

---

# **C# and .NET by Example**

---

Brett L. Schuchert

# Table of Contents

<b>INTRODUCTION</b>	<b>1</b>
<b>WHY DID I START WRITING THIS BOOK?</b>	<b>2</b>
<b>ITERATION 0: JUMP RIGHT IN</b>	<b>3</b>
<b>INITIAL REQUIREMENTS</b>	<b>3</b>
File: Die.cs	3
File: MainApp.cs	3
Build & Execute Instructions	4
Code Explained	4
Initialization	5
Constructor Explained	6
A Memory Snapshot	6
Object Collaboration	7
<b>USING DIE MORE EXTENSIVELY</b>	<b>8</b>
New Requirements	8
A Memory Snapshot	9
Code Explained	10
Build & Execute Instructions	11
Object Collaboration	12
<b>REVIEW</b>	<b>13</b>
Creating classes	13
Attributes	13
Properties	13
Compiling	13
Methods & Constructors	13
Object Creation	13
Sending Messages	14
Arrays	14
Inheritance	14
Using classes in another namespace	14
Output	14
Overloading	14
String Concatenation	14
Overriding	14
Polymorphism	15
<b>ITERATION 1: BOARD GAME</b>	<b>16</b>
<b>A PROBLEM</b>	<b>16</b>
Examine the domain	16
The Static Design	16
Changes from Iteration 0	17
A Memory Snapshot	18
Coding based on dependencies	19
Creating the System	19
Code Explained	21
Object Collaboration	25
<b>REVIEW</b>	<b>26</b>
<b>ITERATION 1A: NO NEW FUNCTIONALITY</b>	<b>28</b>
<b>NAMESPACES</b>	<b>28</b>
A Problem	28
A Solution	28

Build & Execute Instructions	29
<b>PARAMETER CHECKING.....</b>	<b>29</b>
A Problem	29
A Solution	29
Build & Execute Instructions	30
File: Assert.cs	31
<b>COMMAND LINE PARAMETERS.....</b>	<b>33</b>
A Problem	33
A Solution	33
Build & Execute Instructions	33
 <b>ITERATION 2: INHERITANCE &amp; POLYMORPHISM</b>	 <b>35</b>
<b>NEW REQUIREMENTS .....</b>	<b>35</b>
<b>BACKGROUND: INHERITANCE &amp; POLYMORPHISM .....</b>	<b>36</b>
Method and message	36
<b>BUILD 1: A QUICK START.....</b>	<b>37</b>
Updates to: BoardLocation.cs	37
File: StartingLocation.cs	37
Updates to: Board.cs	37
Updates to: Player.cs	38
Updates to: Game.cs	38
Build & Execute Instructions	38
<b>CODE EXPLAINED .....</b>	<b>40</b>
Update: BoardLocation.cs	40
File: StartingLocation.cs	40
Update: Board.cs	42
Update: Player.cs	42
Update: Game.cs	43
Object Collaboration	43
<b>BACKGROUND: STEPS FOR POLYMORPHISM .....</b>	<b>44</b>
<b>BUILD 2: THE REMAINING CLASSES.....</b>	<b>45</b>
File: SurpriseBill.cs	45
File: SpringBoard.cs	45
File: Board.cs	46
Build & Execute Instructions	46
<b>CODE EXPLAINED .....</b>	<b>46</b>
File: SurpriseBill.cs	46
File: SpringBoard.cs	47
File: Board.cs	47
Object Collaboration	48
<b>REVIEW.....</b>	<b>48</b>
Final Observation	50
 <b>ITERATION 2A: READING FROM A FILE</b>	 <b>51</b>
<b>THE PROBLEM.....</b>	<b>51</b>
File: BoardLocations.txt	51
<b>A SOLUTION .....</b>	<b>51</b>
Read lines from a file	52
Parse strings into tokens	52
Constructing the right kinds of objects	53
<b>PUTTING IT ALL TOGETHER.....</b>	<b>54</b>
File: BoardLocationReader.cs	54
Changes to: MainApp.cs	55
File: BoardLocationReader.cs	56
<b>BUILD &amp; EXECUTE INSTRUCTIONS.....</b>	<b>58</b>
<b>USING BOARDLOCATIONREADER IN BOARD.....</b>	<b>58</b>
Update to: Board.cs	58

Update to : Game.cs	59
<b>CODE EXPLAINED .....</b>	<b>59</b>
File: Board.cs	59
File: Game.cs	59
Build & Execute Instructions	60
 <b>ITERATION 2B: REPLACING OUTPUT WITH EVENTS</b>	 <b>61</b>
<b>A PROBLEM .....</b>	<b>61</b>
An Event Standalone Example	61
<b>CODE EXPLAINED .....</b>	<b>63</b>
1. Define an EventArgs class	63
2. Define a delegate	63
3. Define an event sender	63
4. Create an event receiver	64
5. Wire the event receiver to event sender	64
6. Trigger Events	64
<b>A SOLUTION .....</b>	<b>65</b>
Step 1 & 2: TurnEventArgs Class and Delegate	65
Step 3 & 6: Define an Event Sender & Generate Events	66
Step 4: Create an Event Receiver	68
Step 5: Wire Event Listeners	69
Build & Execute Instructions	70
<b>SUMMARY .....</b>	<b>70</b>
 <b>ITERATION 3: ABSTRACT CLASSES</b>	 <b>72</b>
<b>INTRODUCTION .....</b>	<b>72</b>
<b>NEW REQUIREMENTS .....</b>	<b>72</b>
City	72
Transportation	73
Utility	73
Bnb	74
Random	74
<b>BUILD 1: CITY ONLY .....</b>	<b>74</b>
File: City.cs	74
File: BoardLocationReader.cs	76
Build & Execute Instructions	77
Observations	77
<b>BUILD 2: TRANSPORTATION AND UTILITY .....</b>	<b>78</b>
Static Structure	79
Build 2: Code	79
Code Explained	83
Build & Execute Instructions	86
Object Collaboration	87
<b>BUILD 3: OWNING GROUPS OF REAL ESTATE .....</b>	<b>89</b>
Option 1	90
Option 2	91
Option 3	92
Option 4	92
Build 3: Code	92
Build & Execute	99
<b>BUILD 4: A SIMPLE TEXT UI .....</b>	<b>99</b>
Requirements Reviewed	100
Actor to System Interaction	101
Actor to System Design	101
Event Processing	101
Build & Execute	104
<b>BUILD 5: TAKING OUT A LIEN .....</b>	<b>104</b>

Sequence Diagrams	105
Domain Model	107
System Operations/Events	107
Detailed Design Using Collaboration Diagrams	109
Implementing the Design	114
<b>BUILD 6: PLANNING EVENTS</b>	<b>117</b>
<b>BUILD 7: INCREASING CITY TAXES</b>	<b>118</b>
<b>BACKGROUND: OBJECT VISIBILITY</b>	<b>118</b>
Visibility Summary	120
<b>ITERATION 3A: ADO.NET</b>	<b>121</b>
<b>A PROBLEM</b>	<b>121</b>
<b>A SOLUTION</b>	<b>121</b>
Code	121
Code Explained	121
Build & Execute Instructions	121
<b>SUMMARY</b>	<b>121</b>
<b>ITERATION 3B: SIMPLE UI</b>	<b>122</b>
<b>A PROBLEM</b>	<b>122</b>
A Standalone Example?	122
<b>A SOLUTION</b>	<b>122</b>
<b>SUMMARY</b>	<b>122</b>
<b>ITERATION 4: REMAINING CLASSES</b>	<b>123</b>
<b>A PROBLEM</b>	<b>123</b>
<b>A SOLUTION</b>	<b>123</b>
<b>SUMMARY</b>	<b>123</b>
<b>APPENDIX: VISUAL NOTATION</b>	<b>124</b>
<b>STATIC NOTATION</b>	<b>124</b>
The Class	124
Association	124
Multiplicity	124
Role	125
Navigable	125
Dependency	125
<b>DYNAMIC NOTATION</b>	<b>125</b>
Collaboration Diagram	126
<b>APPENDIX: PRE VERSUS POST INCREMENT</b>	<b>128</b>
<b>OVERVIEW</b>	<b>128</b>
<b>IT IS THE SAME</b>	<b>128</b>
<b>WHAT IS THE DIFFERENCE</b>	<b>128</b>
Pre Increment as a Function	128
Post Increment as a function	128
Conclusion	128
<b>APPENDIX: VIRTUAL METHODS EXPLAINED</b>	<b>129</b>
<b>FOLLOW THE CHAIN</b>	<b>130</b>
Object → Class	130
Class → Virtual Method Table	130
Virtual Method Table → Code	131
Lookup	131

ToSTRING..... 131

LANDON..... 132

**APPENDIX: SUMMARY OF THE 5 + 1 LAYERS PATTERN** **134**

PRESENTATION..... 135

APPLICATION ..... 136

BUSINESS..... 136

INTEGRATION ..... 137

EXTERNAL RESOURCES ..... 137

DATA STRUCTURES..... 138

    Exceptions across Layers 138

**REFERENCES** **139**

## Table of Figures

Figure 1: UML – Structure: A Simple Die .....	3
Figure 2: Logical Memory Snapshot .....	7
Figure 3: UML – Collaboration: MainApp.Main().....	7
Figure 4: UML – Collaboration: Explained.....	8
Figure 5: UML – Structure: Die Version 2.....	8
Figure 6: Logical Memory Snapshot: d → PairOfDice.....	10
Figure 7: UML – Collaboration: MainApp → PairOfDice .....	12
Figure 8: UML – Collaboration: PairOfDice.Roll() .....	12
Figure 9: UML – Collaboration: PairOfDice.Value .....	12
Figure 10: A Simple Board Game .....	16
Figure 11: UML – Structure: The structure of one solution .....	17
Figure 12: What our objects look like in memory .....	18
Figure 13: UML – Collaboration: MainApp → Game .....	25
Figure 14: UML – Collaboration: Game.Create(int) .....	25
Figure 15: UML – Collaboration: Board.Create() .....	25
Figure 16: UML – Collaboration: Game.Play(int) .....	26
Figure 17: UML – Collaboration: Player.TakeATurn(PairOfDice) .....	26
Figure 18: UML – Structure: Board Location Hierarchy for Iteration 2 .....	36
Figure 19: UML – Collaboration: Player.TakeATurn(PairOfDice) .....	44
Figure 20: UML – Collaboration: StartingLocation.LandOn(Player)/PassOver(Player) .....	44
Figure 21: UML – Collaboration: SpringBoard.LandOn(Player)/PassOver(Player) .....	48
Figure 22: UML – Collaboration: SurpriseBill.LandOn(Player).....	48
Figure 23: UML – Structure: RealEstate sub hierarchy.....	79
Figure 24: UML – Collaboration: Game.Create .....	88
Figure 25: UML – Collaboration: RealEstate.OfferMyselfForSaleTo .....	88
Figure 26: UML – Collaboration: RealEstate.CalculateAndChargeTaxFor.....	89
Figure 27: UML – Collaboration: Utility.Tax .....	89
Figure 28: UML – Structure: RealEstate & RealEstateGroup.....	90
Figure 29: UML– Collaboration: City.Tax.....	90
Figure 30: UML – Collaboration: Transportation.Tax .....	91
Figure 31: UML – Collaboration: Utility.Tax .....	91
Figure 32: UML – Structure: Parallel Hierarchy .....	91
Figure 33: UML – Structure: Tax Algorithm .....	92
Figure 34: UML – Sequence: The Game in Detail .....	99
Figure 35: UML – Sequence: Actor → System Level Detail .....	100
Figure 36: UML – Sequence: A Turn .....	101
Figure 37: UML – Collaboration: RealEstate.OfferMyselfForSaleTo(p: Player).....	101
Figure 38: UML – Collaboration: TextUI.DisplayTurnEvent(Object, TurnEventArgs).....	102
Figure 39: UML – Sequence: Taking out a lien .....	105
Figure 40: UML – Sequence: Paying back a lien .....	106
Figure 41: UML – Sequence: Outstanding lien overdue .....	106
Figure 42: UML – Sequence: Insufficient funds .....	106
Figure 43: UML – Static Structure: Partial Domain Model .....	107
Figure 44: UML – Collaboration: Choose begin turn actions .....	109
Figure 45: UML – Collaboration: TakeOutALien(Player) .....	110
Figure 46: UML – Collaboration: PayBackALien(Player).....	111
Figure 47: UML – Collaboration: LienOverdue(RealEstate) .....	111
Figure 48: UML – Collaboration: RealEstate.ReleaseLien .....	112

Figure 49: UML – Collaboration: InsufficientFunds(Player, int).....	112
Figure 50: UML – Collaboration: Bankrupt.....	113
Figure 51: UML - Collaboration: Lien.HandleEvent .....	113
Figure 52: A Class .....	124
Figure 53: Associations.....	124
Figure 54: Multiplicity.....	125
Figure 55: Navigable.....	125
Figure 56: Dependency.....	125
Figure 57: Collaboration: Example.....	126
Figure 58: Collaboration: Instances and Classes .....	126
Figure 59: UML – Structure: BoardLocation Hierarchy .....	129
Figure 60: Virtual Function Tables.....	130
Figure 61: Object → Class.....	130
Figure 62: Class → Virtual Method Table .....	130
Figure 63: Virtual Method Table → Code.....	131
Figure 64: Player Passes Over StartingLocation .....	132
Figure 65: Player Lands On a BoardLocation .....	132
Figure 66: 5 + 1 Layer View.....	135



# Introduction

If you are looking to get right in to C#, skip this section. It's here because books tend to start with an introduction of sorts. If you have some spare time you might consider reading it. Otherwise, I recommend going right to "Iteration 0: Jump Right In", starting on page 3.

When you write a program the most important thing you develop is the code. If you do not have code, you have nothing to show for your work. On the other hand, if you try to fix systemic problems by just looking at the code and not looking at the overarching structure, you will not get very far. The way I normally say this is "Code is the most important thing you create and the least point of leverage in a system".

When I learn a new language or technology, I work best from concrete examples. Show me how to do it at first, give me some amount of success and then I'll be able to build on that success by reading help files, experimenting, asking other people. The most important thing is for me to get something working. There's an old saying, get it to work, get it to work right, get it to work fast. I don't actually prefer this version but it suffices. I'd recommend replacing "fast" with "well" and allow the definition of "well" to vary depending on the context.

What is this book going to do for you? On the one hand there is a lot of code. You will have several concrete examples of using C# in ways you would actually use it on a project. You will work through a single project and modify it over many function-adding iterations, many non-function adding iterations and several intermediate steps, or builds, within some of the iterations. This book will cover only certain kinds of projects, which may limit its usefulness but you need to learn to walk before you learn to run. The second volume in the series will take this approach to its logical conclusion by continuing where this book leaves off but with a whole new project and addressing enterprise level application development.

This book will also show you how to think about approaching problems somewhat formally. Not so much that the notation or documentation gets in the way. However, it is formal enough that if you need to have a big impact on the development of a system you will have enough to get started. The diagrams use UML almost exclusively for the notation. The approach is similar to a lightweight version of the Unified Process. So you will better understand UML and the Unified Process.

This book uses C# as the programming language and .NET as the framework. You really cannot discuss one without the other. (This is not strictly true, you can talk about .NET without C# but you'd have to use some other programming language such as VB.NET so I'm not lying to you too much.)

What is this book not? This book does not fully cover C#, I don't want it to. There are several excellent C# language reference manuals, writing another would simply add to the quickly-rising noise. This book does not fully cover .NET, it's not even close. As I write this I barely understand .NET. There are several books that cover various part of the .NET framework and the documentation that comes with the Microsoft .NET SDK is both large and fairly comprehensive.

How should you use this book? The Sapir-Whorff hypothesis states:

## *Da dad a da*

To interpret this, you need to have the language to understand the thing. This provides something of an initial value problem. We need to know C# to be able to use C#. How do you make it over the initial hurdle? Follow examples, reflect, learn a little, and learn a bit more. Eventually you will be able to speak C#. Knowing C# without knowing .NET is mostly useless. Much of the power of C# is not in the language but in what you get when using .NET. You can think of .NET as a larger vocabulary. The larger your vocabulary, the more you will be able to clearly and succinctly express your thoughts. For example, you could express your thoughts on reading a file by writing your own library to read files. If you do this, anybody who wants to understand you will have to understand your library. If you and 1000 other people do the same thing, then it makes communication almost impossible. .NET

gives us a way to read a file. If we both learn how to do it using .NET, then we can speak about reading a file using .NET and quickly discuss a very complex topic with very few words.

Is knowing C# and .NET enough? No. You need to know how to solve problems with these. We can start by writing a bunch of code to solve a problem. We might even be successful but eventually this tried and true method of development reaches the limits of its effectiveness. We need to be able to approach problems somewhat formally so we can solve bigger and more complex problems.

## Why did I start writing this book?

I started working with C# while working in England during the summer of 2002. Like Java, I started with a standard problem to learn how to do certain basic things in the language. Along the way I also did a few other side things with this project and decided that it might be a good article. I started writing an article and it quickly became too long. I'd sneak in an hour here; a few hours there and all of a sudden what I thought might be a series of articles turned into what I hoped would be a book.

I might have tried to write a book about designing Object-Oriented systems in C# and .NET. That would be a good book. However, the way I practice Object-Oriented is mostly language neutral. In reality when I approach a problem I do not really concern myself too much with the programming language. It is essential to be able to express thoughts in a language. It is also the least influential place to do it. I think that is an interesting dichotomy.

You need to be able to think about the problem without concerning yourself with the language. Once you can do this, you have a foundation upon which to learn how to think about design. After you are comfortable with design, analysis becomes possible. After that, maybe capturing requirements, working on system's engineering issues, architecture, etc. It all starts with a language.

The Sapir-Whorf hypothesis states that you do not really understand a thing until you have a language in which to express it. Thought follows language. Contemporary theories of cognitive science take this even further. So this book first and foremost is about learning the language to express solutions, where the language happens to be C# and .NET. Along the way we will learn how to implement several basic building blocks, including inheritance and polymorphism. We will learn a visual language, called UML, for expressing ideas and relationships between things. We will learn how to do things in the way that .NET would prefer you do them.

The approach, I hope, is simple. Give you a problem description and a basic design and give you the code to solve the problem. You manually type in the code, compile it, get it to run then read about the parts of the code I thought were worthy of note. Every so often we take little sidebars that do not necessarily increase the functionality of the solution but do make it better in some way.

To do this, we use one problem. This problem is based on the Monopoly® Board Game, although other than this reference to the game, we do not use any of the names, and we change the rules to make the problem a different solve.

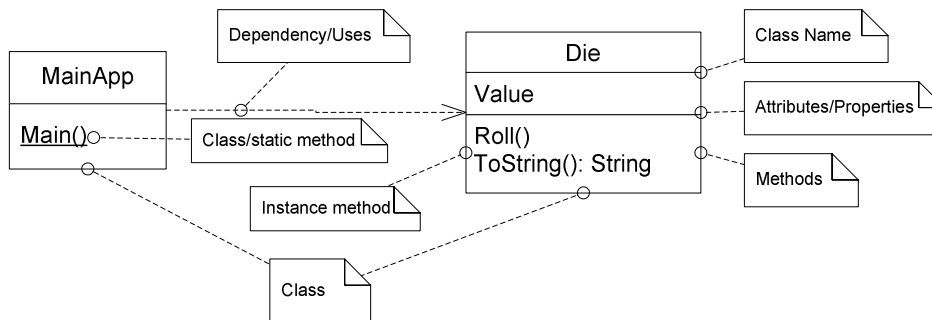
Using this problem has a long history. In a previous life I worked at a place where we trained people and consulted. (I currently work at a similar place.) One of the founders used this problem once in a C++ class. It turned out to be an excellent problem to cover basic things like creating classes, object interaction, encapsulation, inheritance and polymorphism. Since then when I'm working with a new language, this is the first problem I write. I've written part of this game probably more than 100 times. It might be a game but I am able to relate difficult design problems back to ideas in Monopoly® better than 70% of the time, so I love this problem.

Feel free to contact me about anything in this book. My personal email is [Schuchert@yahoo.com](mailto:Schuchert@yahoo.com).

# Iteration 0: Jump Right In

## Initial Requirements

For our first class, imagine you want to be able to roll dice. Imagine a simple set of classes that look something like the following:



**Figure 1: UML – Structure: A Simple Die**

In C# we'd need to write some simple code to create this class. To define the Die class we might write the following:

### File: Die.cs

```

1| using System; // we use the Random class
2| public class Die {
3|     private const int SIDES = 6;
4|     private static Random RNG = new Random();
5|
6|     private int value;
7|
8|     public int Value {
9|         get { return value; }
10|     } // end Value property
11|
12|     public void Roll() {
13|         value = RNG.Next(SIDES) + 1;
14|     } // end Roll method
15|
16|     public override string ToString() {
17|         return "Die(" + Value + ")";
18|     } // end ToString method
19| } // end Die class
  
```

This just defines a class. We have not written any code to use the class. The following example uses the Die class:

### File: MainApp.cs

```

1| using System; // we use the Console class
2| public class MainApp {
3|     public static void Main() {
4|         Die d = new Die();
5|         Console.WriteLine("Initial Value: {0}", d.Value);
6|         d.Roll();
7|         Console.WriteLine("Value Rolled: {0}", d.Value);
8|         Console.WriteLine(d);
9|     } // end Main method
10| } // end MainApp class
  
```

## Build & Execute Instructions

To build these two files, do the following:

1. Create these two files using the names provided. Save both files in the same directory<sup>1</sup>.

2. Start a command prompt and issue the following command:

```
c:\C#\BoardGame>csc *.cs
```

3. Execute your program with the following command:

```
c:\C#\BoardGame>MainApp
```

4. You will see something like the following appear:

```
Initial Value: 0
Value Rolled: 3
Die(3)
```

## Code Explained

As we work through examples we will reveal more about the code. For now we are lying to adults.

### File: MainApp.cs

#	Description
1	<i>Using System;</i> Classes exist in namespaces. Tell C# that we want to make the names in the System namespace available without having to fully qualify the name. This allows line 6 to refer to Console instead of System.Console.
2	<i>Public class MainApp {</i> Define a class called MainApp. This class is public, meaning it is available to other classes regardless of the locations of those classes.
3	<i>Public static void Main() {</i> Define a method called Main(). Main() is the entry point for any C# system. Main() must be public so it can be called from elsewhere. Main() must be static because it is called before any instances of the class (MainApp in this case) exist. For now realize that this method must always be public and static.
4	<i>Die d = new Die();</i> Declare a reference to a Die object called d. Initialize with a newly created Die object.
5	<i>Console.WriteLine("Initial Value: {0}", d.Value);</i> Display the message "Initial Value: n" where n is filled in with the current value of the Die. Since we have not called roll, this value is set to whatever C# uses for default initialization. C# initializes the attribute to 0 so we will see 0.
6	<i>d.Roll();</i> Send the Die object called d the message called Roll().
7	<i>Console.WriteLine("Value Rolled: {0}", d.Value);</i> Display the message: "Value Rolled: n" where n is filled in with the current value of the Die.
8	<i>Console.WriteLine(d);</i> Display the Die object called d using the WriteLine method of Console.

### File: Die.cs

#	Description
1	<i>using System;</i> Allow line 5 to refer to the class Random without calling it System.Random.
2	<i>public class Die {</i> Define a class called Die (the single form of Dice). This class is public, which makes it available to other classes regardless of the locations of those classes.

<sup>1</sup> For this example we'll assume the directory is called c:\C#\BoardGame.

#	Description
3	<code>private const int SIDES = 6;</code> Define a constant called SIDES. This represents the number of sides of the die. We define a constant here and use in later on in line 13. This avoids the use of so-called magic constants.
4.	<code>private static Random RNG = new Random();</code> Define a class variable called RNG. It is an instance of the Random class. The Random class generates random values. We have made this private so it can only be used by methods in the Die class. It is static so that there will only be one of these no matter how many Die objects exist. We initialize it here so we can use it later and not worry about it. We actually use this object in the Roll() method on line 13.
6	<code>private int value;</code> Define an attribute. Instances of the Die class have a value. Each object of the class Die has its own copy of this value.
8	<code>public int Value {</code> Begin the definition of a public property. A property describes a characteristic of an object. That characteristic might be directly associated with an attribute, or it might also be calculated. In this case our Value property directly uses the value attribute. Later we will see calculations using attributes. The MainApp.cs program uses this property on line 6. The Die.cs program uses this on line 16 in the ToString method.
9	<code>get { return value; }</code> This is the code to get the Property. Since there is no corresponding set property, this is a read-only property.
12	<code>public void Roll() {</code> Define a public method called Roll. This is the message sent to change the value of the die from its current value to some new random value.
13	<code>value = RNG.Next(SIDES) + 1;</code> To create a random value between 1 and 6, we send RNG, an instance of the Random class, the message Next and pass in SIDES. The SIDES constant is equal to 6. The Next method returns back a new random number between 0 and 5. We add 1 to make the range 1 to 6. We then assign this number to the value attribute.
16	<code>public override string ToString() {</code> Override the method called ToString. This method returns a string that represents the object. All classes in C# either directly or indirectly inherit from the base class Object. It defines a method called ToString. The ToString method is called when an object is passed into Console.WriteLine, among other places. By writing this method we get nice output from line 7 of MainApp.cs.  We must use the override keyword to indicate to C# that we mean to be replacing the method defined in the Object class with our own version.
17	<code>return "Die(" + Value + ")";</code> Create a new string by concatenating the literal "Die(" with the Value property of the class and adding a ")". The result is a string in the form Die(n).

## Initialization

Just after creating the Die object we display its current value, which is 0. C# automatically initializes the value attribute to 0. Our Die object, however, should never have a value of 0. Its range of valid values is 1 through 6. We need to fix the Die class so that when someone uses it, it will properly initialize itself. We have a few options.

- Just set the value to 1 or some other reasonable default value.
- Set the initial value to a random value between 1 and 6.

To perform automatic, guaranteed initialization, we add one or more constructors to a class. Regardless of which of the above two options we choose a constructor will perform initialization.

We chose the second option. To make this happen, add the following method to your Die class:

#### Changes: Die.cs

```
1| // add the following constructor
2| public Die() {
3|     Roll();
4| } // end Die constructor
```

#### Build & Execute Instructions

Rebuild your system and notice the change in the output:

1. Update Die.cs by adding the constructor.

2. Compile your system:

```
c:\C#\BoardGame>csc *.cs
```

3. Execute your program with the following command:

```
c:\C#\BoardGame>MainApp
```

4. You will see something like the following appear:

```
Initial Value: 2
Value Rolled: 3
Die(3)
```

#### Constructor Explained

#	Description
2	<code>public Die() {</code> Define a constructor for the Die class. A constructor must have the same name as the class. This constructor takes no parameters.
3	<code>Roll();</code> Call the Roll() method. Since this method sets the value attribute in the range 1 to 6, after the constructor finishes our Die object is in a well-defined state.

The system executes a constructor for every object your code creates. When we write:

```
Die d = new Die();
```

C# automatically allocates enough memory to hold the Die object. It next calls a constructor. Our constructor uses the Roll() method to randomize the value.

#### A Memory Snapshot

When we write code to create objects, C# automatically allocates memory for those objects in a place of memory called the heap. A program's memory can be split into three logical areas:

- **Heap** – The place where the system allocates memory on demand for objects.
- **Stack** – The place in memory the system uses to pass parameters to functions, store local variables and remember where to return after calling functions.
- **Code** – The place in memory where the system stores the program code and constants.

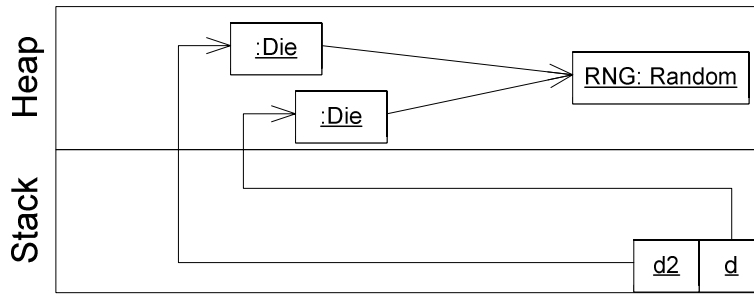
In practice memory is more complex than this but this model will suffice for the next discussion.

After line 4 of MainApp.cs, we have a reference called d on the stack. We also have an instance of the Die class residing on the heap. The reference d currently refers to that instance on the heap. In addition, all instance of the Die class share a single instance of the Random class, which is called RNG.

Imagine we add another line after line 4:

```
Die d2 = new Die();
```

After creating two these two Die objects, the stack and heap logically resemble the following picture:



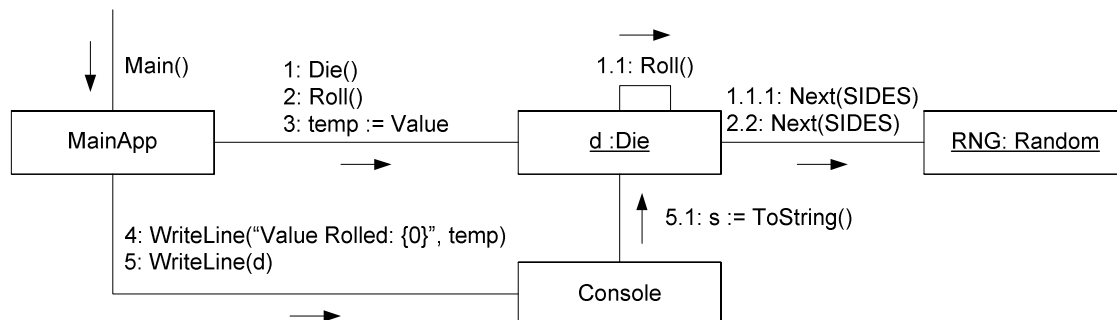
**Figure 2: Logical Memory Snapshot**

The above diagram shows that both Die objects on the heap refer to the same random object. Conceptually this is correct. In practice, C# takes care of this reference for us. If we were to look at the actual bytes of memory representing the two Die objects, we would not actually see the reference to the Random object. We would have to look in the code of the Roll() method to see the actual reference to the Random object. This is probably too much detail to worry about, thus the diagram is called a logical memory snapshot.

The order of d and d2 is deliberate. Stacks are called stacks because they grow and shrink at one end, like a stack of plates. In fact they tend to grow from the end down. As we create more references in a method, the references go on the stack. This is also logical and in practice not terribly important to know. With older languages such as C++ this can be important because it is possible to accidentally “step over” other objects in memory. In C# such mistakes are, while not impossible, quite difficult to make.

## Object Collaboration

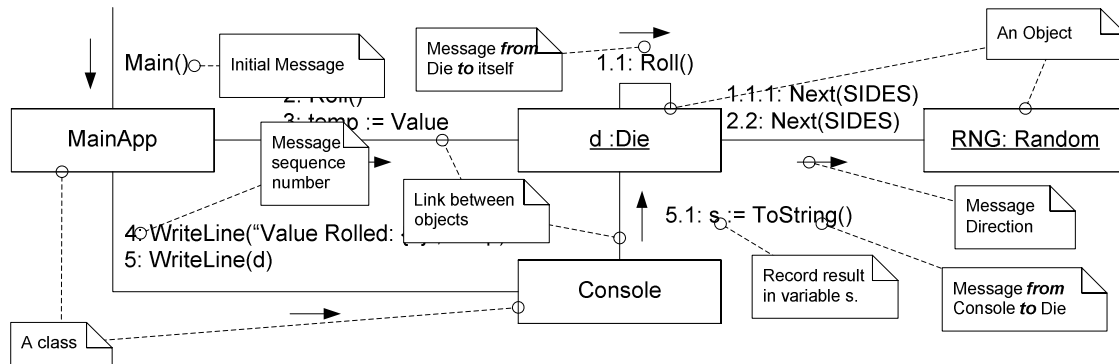
Before moving on to the next coding exercise, we will review collaboration between the objects and classes in our system. The following diagram shows the interactions between objects and classes starting with the Main() method in the MainApp class:



**Figure 3: UML – Collaboration: MainApp.Main()**

This is the first collaboration diagram. While we describe these in detail in “Appendix: Visual Notation” starting on page 124, we provide this simple explanation so you do not need to flip back to the appendix or now.



**Figure 4: UML – Collaboration: Explained**

When you execute the program, the system will send the Main() message to the MainApp class. The MainApp.Main() method first creates a Die object, which it stores in the reference named d. It next sends the Roll() message to d. On line 6, the Main method acquires the Value property of d just before sending the WriteLine message to the Console. Finally, Main sends Console a different WriteLine message.

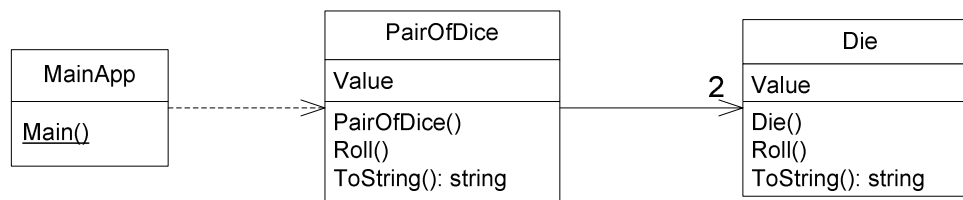
When Main() creates the Die, its constructor is automatically called. The constructor sends itself the Roll() message, which guarantees that the current value of the Die object is a reasonable value. By reasonable we mean a value between 1 and 6. Its default value of 0 violates the semantics of a die object; when was the last time you picked up a 6 sided die object and saw a 0?

There are two places from which the Die object receives the Roll message. The first is from within its own constructor; the second is from the Main method. In both cases the same thing happens, it sends the Random object the Next message to get another random value, which it stores in its attribute called current.

## Using Die more extensively

### New Requirements

In this next step we create a class called PairOfDice that holds on to two die objects. It will be able to Roll() both die objects for us and return the sum of the two die objects. We will be using this class later on so we'll build it now. Here's how PairOfDice relate to Die:

**Figure 5: UML – Structure: Die Version 2**

### PairOfDice.cs

```

1 | public class PairOfDice {
2 |     private const int DICE_COUNT = 2;
3 |     private Die[] dice;
4 |
5 |     public PairOfDice() {
6 |         dice = new Die[DICE_COUNT];
7 |         for(int i = 0; i < DICE_COUNT; ++i) {
8 |             dice[i] = new Die();
9 |         } // end for loop
10 |    } // end PairOfDice constructor
11 |
12 |    public void Roll() {
  
```



```
13|         foreach(Die d in dice) {
14|             d.Roll();
15|         } // end of foreach loop
16|     } // end of Roll() method
17|
18|     public int Value {
19|         get {
20|             int sum = 0;
21|             foreach(Die d in dice) {
22|                 sum += d.Value;
23|             } // end of foreach loop
24|             return sum;
25|         } // end of get section
26|     } // end of Value property
27|
28|     public override string ToString() {
29|         string rep = "PairOfDice(";
30|         bool first = true;
31|         foreach(Die d in dice) {
32|             if(first == true) {
33|                 first = false;
34|             } else {
35|                 rep += ", ";
36|             } // end of if
37|             rep += d.Value;
38|         } // end of foreach loop
39|         rep += ")";
40|
41|         return rep;
42|     } // end of ToString method
43| } // end of PairOfDice class
```

Update the MainApp to use the PairOfDice class instead of the Die class:

#### File: MainApp.cs

```
1| using System;
2| public class MainApp {
3|     public static void Main() {
4|         PairOfDice d = new PairOfDice();
5|         Console.WriteLine("Initial Value: {0}", d.Value);
6|         d.Roll();
7|         Console.WriteLine("Value Rolled: {0}", d.Value);
8|         Console.WriteLine(d);
9|     } // end of Main() method
10| } // end of MainApp class
```

### A Memory Snapshot

After line 4, MainApp has one local variable, d, which is a reference of type PairOfDice. The reference points to an instance of PairOfDice on the heap. A PairOfDice instance has a single attribute, an array. The array refers to two die objects, both on the heap. Conceptually every Die object shares a single reference to a Random object, which is known as RNG:

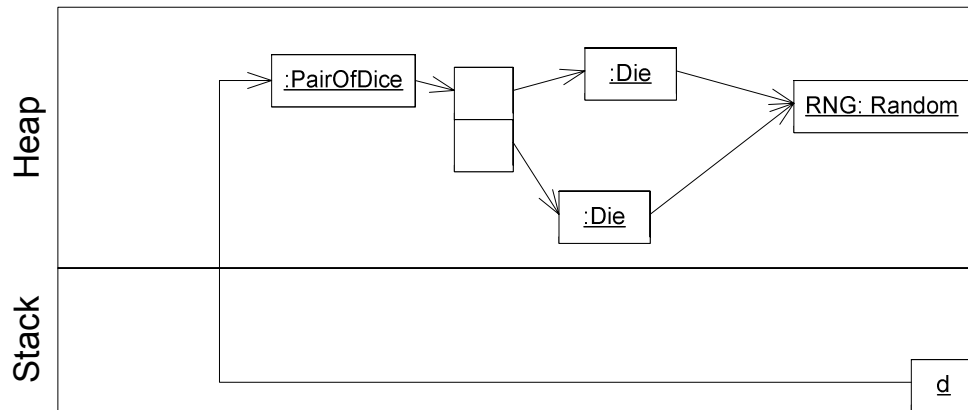


Figure 6: Logical Memory Snapshot: d → PairOfDice

### Code Explained

File: PairOfDice.cs

#	Description
1	<code>public class PairOfDice {</code> Define a new class called PairOfDice. This class is public, making it universally available.
2	<code>private const int DICE_COUNT = 2;</code> Define a constant. This constant represents the number of die objects each instance of the PairOfDice will have.
3	<code>private Die[] dice;</code> Declare an array that will hold on to die objects. The name of the array, dice, is a reference to the array. It is null until we initialize it in the constructor on line 6.
5	<code>public PairOfDice() {</code> Declare a constructor. This constructor is public so it can be used anywhere. Furthermore, it takes no parameters so it is called a default constructor.
6	<code>Dice = new Die[DICE_COUNT];</code> Allocate an array of references to die objects of size 2. We still have not created any die objects, we now have an array which is capable of holding on to die objects.
7	<code>for(int i = 0; i &lt; DICE_COUNT; ++i) {</code> Loop from 0 to 1. This is the loop where we finally create die objects. We cannot use foreach, which we will see below, because the array only contains null references. In addition, if you have seen loops like this before, you are probably use to seeing i++ instead of ++i. If you are interested in knowing why these examples use the pre-increment form, see “Appendix: Pre versus Post Increment” starting on page 128.
8	<code>Dice[i] = new Die();</code> Allocate a new Die object and store it in the array.
12	<code>public void Roll() {</code> Define a public method called Roll. This looks just like the method in Die. The difference is this works with two die objects where the Roll method in the Die class just works on the one die object; the object the message was sent to.
13	<code>foreach(Die d in dice) {</code> The foreach keyword is a way to iterate through all of the elements of a collection or an array. It starts at the beginning and goes through to the end. It enters the block of code immediately following it (the code in the {} just below it) one time for each element in the array. The variable d is first set to dice[0], then dice[1].
14	<code>d.Roll();</code> Send the message Roll() to one of the die objects.

#	Description
18	<code>public int Value {</code> Define a public property called Value. This one is a read-only property because there is only a get section. This shows an example of a complex property.
20	<code>int sum = 0;</code> We are going to sum up all of the current values of the dice in the array. Start with a sum of 0.
21	<code>foreach(Die d in dice) {</code> Iterate through each of the die objects in the dice array.
22	<code>sum += d.Value;</code> Add the Value of a given die to the sum.
24	<code>return sum;</code> Return the sum as the last step in the property.
25	<code>public override string ToString() {</code> Define another ToString method for ease of printing.

**File: MainApp.cs**

#	Description
4	<code>PairOfDice d = new PairOfDice();</code> Declare a variable, d, which has the reference type PairOfDice. This is the only change from the original version. Instead of using the Die class, the code uses the PairOfDice class.
6	<code>d.Roll();</code> The system knows to use the Roll() message in the PairOfDice class because the reference type of d is PairOfDice and not Die.

**Build & Execute Instructions**

To build these three files, do the following:

1. Create PairOfDice.cs, update MainApp.cs and make sure all three files are in the same directory.

2. Start a command prompt and issue the following command:

```
c:\C#\BoardGame>csc *.cs
```

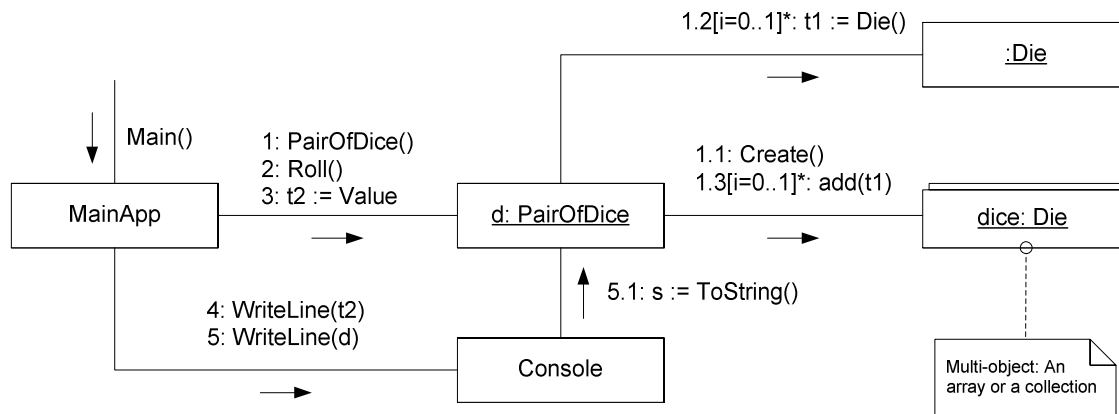
3. Execute your program with the following command:

```
c:\C#\BoardGame>MainApp
```

4. You will see something like the following appear:

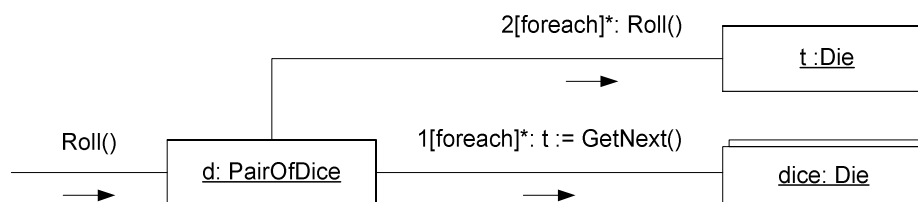
```
Initial Value: 5
Value Rolled: 9
PairOfDice(3, 6)
```

## Object Collaboration



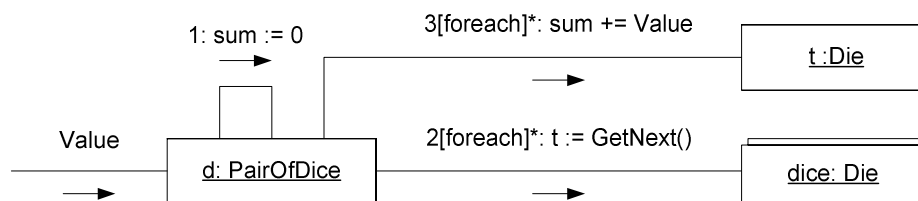
**Figure 7: UML – Collaboration: MainApp → PairOfDice**

The only change to `MainApp.Main` is that it creates an instance of `PairOfDice` instead of an instance of `Die`. What happens as a result is more complex. When we create a `PairOfDice`, it first creates an array, shown above as a multi-object. We then use a loop to create new `Die` instances and add them into the array. We use a generic notation for both looping and collections so that we do not have to show special notation for Collection classes versus Array objects.



**Figure 8: UML – Collaboration: PairOfDice.Roll()**

When `MainApp.Main` sends the `Roll` message to the `PairOfDice`, the result is somewhat more complex than `Die.Roll`. The `PairOfDice` uses `foreach` to iterate through its array of `Die` objects and sends each on the `Roll` message.



**Figure 9: UML – Collaboration: PairOfDice.Value**

The collaboration for `Value` is much the same as `Roll`. The differences are that we initialize a `sum`, we use the `Value` property instead of the `Roll` message to each `Die` object and we add the value returned from the `Value` property into the `sum`.

## Review

So far we have created three classes. Die and PairOfDice are two classes that might appear in something like a game. MainApp is a class we've created so we have some place to put a Main() method. What have we accomplished so far:

### Creating classes

- The following is a minimal class: `class AClass{ }`
- We added public in the front to make the classes accessible to other classes. Since we are not yet too worried about working between assemblies, this is not a terribly important point.

### Attributes

- Die and PairOfDice both have an attribute. Die's attribute is a simple int. PairOfDice is complex in two respects, first it is an array, second it holds on to instances of a class we defined previously.

### Properties

- A property is like an attribute. It is exposed to users of the class. It is accessed without using (), which makes it look different from sending a message.
- A property can be read only. It can be read/write or even write only. So far we have not needed to write to a property.
- Properties have code associated with them. The code might be simple, as in the case of Die, or complex, as in the case of PairOfDice.

### Compiling

- We used csc from the .NET SDK. It compiles source files and generates MSIL.
- Compiling a group of files with csc generates a single executable with no other intermediate files.
- The name of the generated executable is the same as the name of the class that contains the Main() method. We ran the program called MainApp.exe.

### Methods & Constructors

- Classes have methods. Constructors are a special kind of method automatically called when we create an instance of a class.
- A constructor has the same name as the name of the class. It cannot have a return type.
- A method has a name, must have a return type, even if it is void, and might take parameters.

### Object Creation

- To create an object, we use new.
- Using new does three things:
  - It allocates memory on the heap to hold the object.
  - It calls the constructor to initialize the memory.
  - It returns a reference to the newly allocated and initialized object
- We do not have to worry about getting rid of objects; C# performs garbage collection for us. Garbage collection is the means by which the system removes objects that are no longer in use.

## **Sending Messages**

- The syntax for sending a message is: `object.Message()`. The capitalization is only important in that it must be the same where we use it as where it was defined (not so for Visual Basic.NET).

## **Arrays**

- To declare an array, use `[]`. An example: `int arrayOfInt[];`
- To allocate memory for an array, use `new`: `arrayOfInt = new int[10];`
- To access an array, use `[]`: `arrayOfInt[3]`.
- To set a value in an array use `[]`: `arrayOfInt[5] = 1967.`
- The valid indexes for an array are 0 to (size – 1). An array of size 4 has the following valid indexes: 0, 1, 2, 3. Arrays are referred to as being 0-based in C#.

## **Inheritance**

- All classes inherit either directly or indirectly from the `Object` class.
- The `Object` class has some methods we might want to override.

## **Using classes in another namespace**

- The `using` keyword allows us to refer to a class by a shorter name. Instead of using a fully-qualified name to perform output, we used a shorter name. That is, we used `Console.WriteLine(...)` instead of `System.Console.WriteLine(...)`;

## **Output**

- The `System.Console` class allows us to perform simple output.
- We used the `WriteLine` method of the `System.Console` class to write output and include a new line in the output.

## **Overloading**

- We used two different methods with the same name but different parameters when writing output.
  - `Console.WriteLine(d.Value)` writes an integer because the expression `d.Value` returns an integer.
  - `Console.WriteLine(d)` writes an object because `d` is an object. In this particular case, C# actually converts `d` to a string by sending the `ToString()` message to `d` and printing the results.

## **String Concatenation**

- We can add two strings together using `+`. “Hello” + “World” → HelloWorld.
- We can add a string and a number because the `+` operator is overloaded to handle different combinations.

## **Overriding**

- To replace a method that a super class defines, use the keyword `override`.
- `Object` defines a `ToString()` method. We redefined it in `Die` and `PairOfDice` to do something specific to our classes.

## Polymorphism

When we printed a Die object, the WriteLine() method of the Console class sent the ToString() message to the Die object. The Die class has a ToString() method so the ToString() message invokes the ToString() method of the Die class.

When we printed a PairOfDice object, the WriteLine() method again sent the ToString() message to the PairOfDice object. The PairOfDice class also implements a ToString() method, so the ToString() message invokes the ToString() method in the PairOfDice class.

In both cases, the same method, WriteLine(), sends the same message, ToString(). However, the message, ToString() is sent to two different kinds of objects, that is objects of two different classes. The system determines at runtime the correct version of the ToString() method to execute based on the type or class of the target object. This is a simple example of polymorphism.

## Iteration 1: Board Game

### A Problem

Now we are going to create something more complex. It will use the classes we've already created and it will be something on which we can build in quite a few directions.

Imagine a simple board game. We have a number of players, say between two and four. To keep it simple, nothing happens other than players roll dice and move around a board. Each player takes a turn by rolling a pair of dice and advancing from their current position a number of locations equal to what they rolled. They keep doing this for 10 turns.

We have a simple board game with players rolling dice and moving around the board:



**Figure 10: A Simple Board Game**

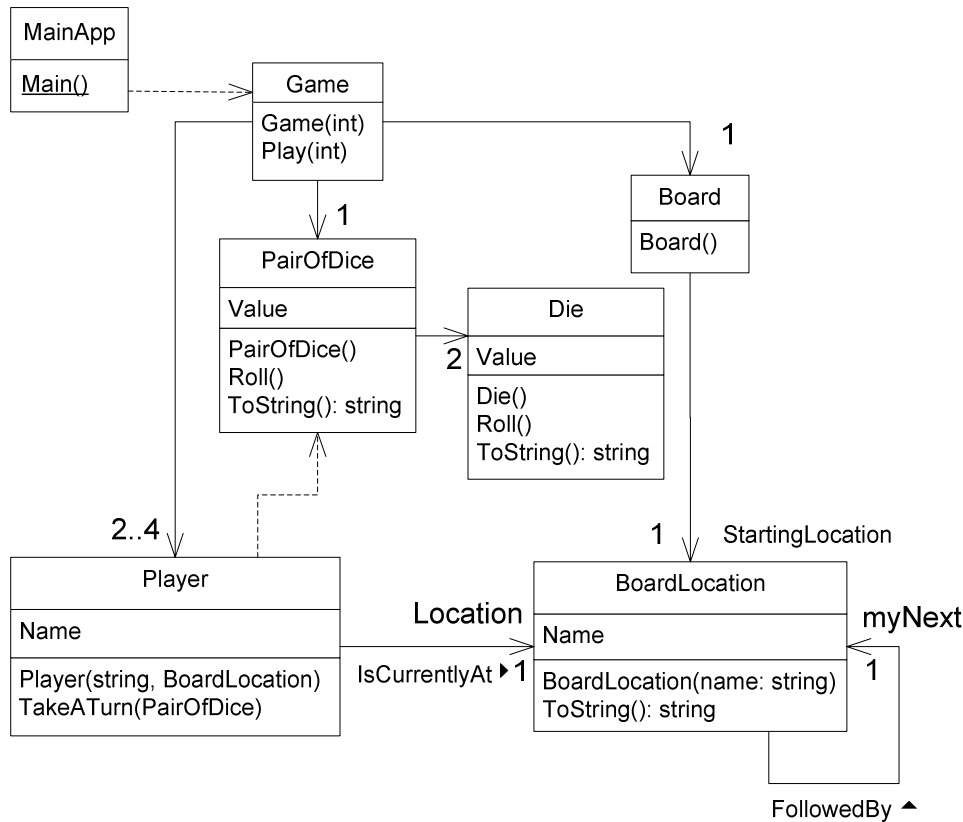
### Examine the domain

If we examine the problem statement of the picture, this problem seems to have several building blocks. These include player, board, board location, game, die. We might spend time using things like the GRASP [4] in an attempt to design a set of classes to solve this problem. Making design decisions is currently out of what we're trying to accomplish. We begin to address making design decisions later in the book. For now we move ahead with a design that happens to work well for where we will be taking this problem

### The Static Design

A static design represents the parts of the solution along with how those parts relate. If we consider the key building blocks and how those relate, we might come up with the following diagram:





**Figure 11: UML – Structure: The structure of one solution**

This is quite a bit more complex than the previous examples. We will first take a quick look at each of these classes. Next, we will examine what this system logically looks like when it is running. Finally, we present the source code, commentary and a summary.

## Changes from Iteration 0

### PairOfDice, and Die

These classes are unchanged. We are just using them in their current form.

### MainApp

MainApp will have a Main() method. Instead of using a Die or a PairOfDice, it uses the Game class. It will create one Game object and then send it the message Play().

### Board Location

Players move around the board. They start on one location and end up on another. It is the place where players reside between turns. A Board Location knows its Name.

There is a solid line with an arrow from a Board Location to itself. This means that one instance of the Board Location class can navigate to exactly one other instance of the Board Location class.

### Board

The board is simply a convenient container for all of the Board Locations. Right now it will seem like overkill. Later on (since we know where we are going) it will serve a purpose. We will have it responsible for creating each of the Board locations.

A Board can only directly navigate to one Board Location and that single location is known as the Starting Location. The combination of the Board knowing one Board Location and each Board Location knowing one Board Location gives an unlimited board size.

## Player

When a player plays this game, it first will send the Roll() message to an instance of the PairOfDice class. The player will then ask for the Value to know how many BoardLocation objects forward to advance.

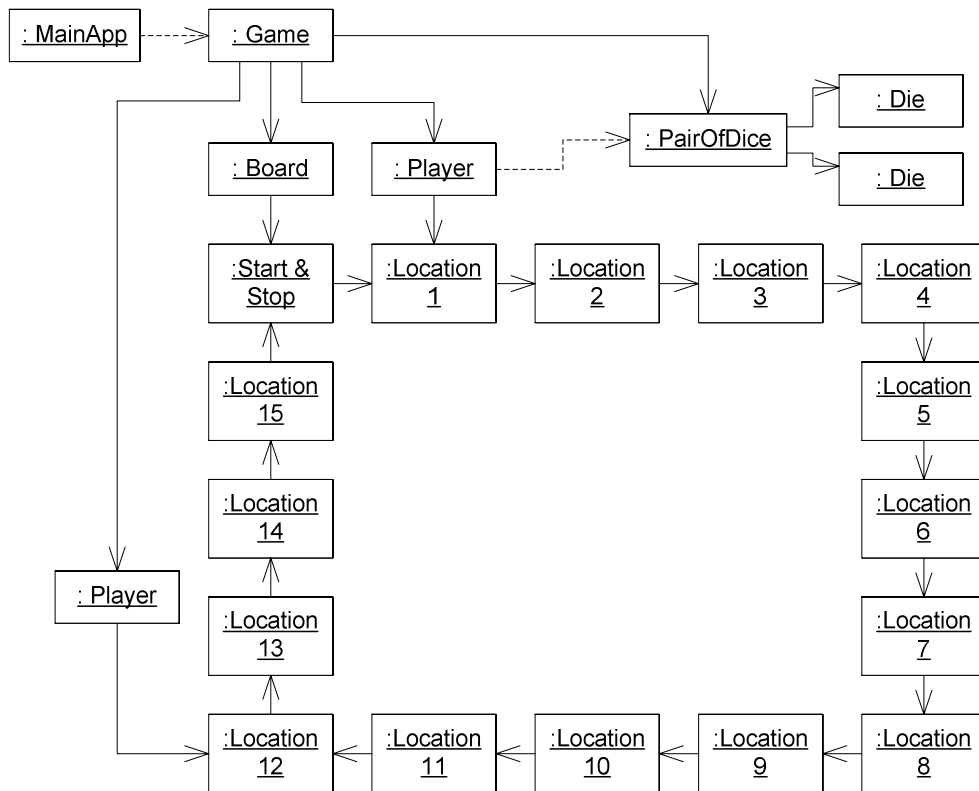
The Player knows its name. The player also knows its location, which it remembers in an attribute, and which is represented as a solid line pointing from Player to BoardLocation. A player uses the PairOfDice as well, shown on the diagram with a dashed line pointing from Player to PairOfDice.

## Game

The Game class pulls everything together. It has one Board, one PairOfDice and between 2 and 4 players.

## A Memory Snapshot

This is what an ongoing game might look like in memory:



**Figure 12: What our objects look like in memory**

Note that in our actual system we will have 40 rather than 16 BoardLocation objects. We show 16 on this diagram to make it small enough to fit on the page but large enough to be representative.

## Observations

- There are a total of 16 Board Locations. We could have only had a few or a few thousand.

- We can support any number of players; in this case there are only two. The requirements provide an example of between 2 and 4 players. In this diagram we have not shown any such limitation.
- One player is currently on Location 1 and another is on Location 12. The player has a reference to an object of the Board Location class. The Player does not have a string or a number identifying the Board Location it is on.
- The player with a dashed line between itself and the PairOfDice is the current player (you only get to Roll() the PairOfDice if it is your turn).
- In all the cases where we have a solid line, our code will show an attribute within a class.
- In all cases where we have a dashed line, our code will show one object using another object in some way. A dashed means we do not have an attribute.

## Coding based on dependencies

The following table summarizes the work to be done to make all of this happen:

Class/File	Notes	Dependencies
MainApp.cs	We will have to update it to use the Game.	Game
Game.cs	We need to create this from scratch.	Board, PairOfDice, Player
Board.cs	We need to create this from scratch.	BoardLocation
PairOfDice.cs	Completed	Die
Die.cs	Completed	None
Player.cs	We need to create this from scratch.	PairOfDice, BoardLocation
BoardLocation.cs	We need to create this from scratch.	None

We can just create all of the missing files and update MainApp.cs and try to get everything to work at the same time. To avoid experiencing too many problems at once, we will instead create the files from least dependent to most dependent. This way we can write some code then compile it and then move on knowing that part of the system builds.

Reviewing the dependencies column above, we see that BoardLocation is the least dependent of the classes not already written. After that we can create either Board or Player. Game comes next and finally we will update MainApp. Notice that what we've done is follow the lines between classes and worked backwards against the arrowheads to figure out which classes are least dependent.

## Creating the System

We recommend creating one file at a time, in the order we have listed the files. After creating each file, compile it using:

```
c:\C#\BoardGame>csc /t:module *.cs
```

If you encounter any compilation errors, fix them before moving on. The /t:module command line option tells C# to build an object module rather than building an executable.

### File: BoardLocation.cs

```

1| public class BoardLocation {
2|     private string myName;
3|     private BoardLocation myNext;
4|
5|     public BoardLocation(string name) {
6|         myName = name;
7|     } // end BoardLocation constructor
8|
9|     public string Name {
10|         get { return myName; }
11|     } // end Name property
12|

```

```
13|     public BoardLocation Next {
14|         get { return myNext; }
15|         set { myNext = value; }
16|     } // end Next property
17|
18|     public override string ToString() {
19|         return "BL(" + Name + ")";
20|     } // end ToString() method
21| } // end BoardLocation class
```

**File: Board.cs**

```
1| public class Board {
2|     private BoardLocation myStart;
3|
4|     public Board() {
5|         myStart = new BoardLocation("Starting Location");
6|
7|         BoardLocation current = StartingLocation;
8|         BoardLocation newNext = null;
9|
10|        for(int i = 1; i < 40; ++i) {
11|            newNext = new BoardLocation("Loc#" + i);
12|            current.Next = newNext;
13|            current = newNext;
14|        } // end for loop
15|        current.Next = myStart;
16|    } // end Board constructor
17|
18|    public BoardLocation StartingLocation {
19|        get { return myStart; }
20|    } // end StartingLocation property
21| } // end Board class
```

**File: Player.cs**

```
1| using System;
2| public class Player {
3|     private string myName;
4|     private BoardLocation myLocation;
5|
6|     public Player(string name, BoardLocation startingLocation) {
7|         this.myName = name;
8|         myLocation = startingLocation;
9|     } // end Player constructor
10|
11|     public string Name {
12|         get { return myName; }
13|     } // end Name property
14|
15|     public BoardLocation Location {
16|         get { return myLocation; }
17|     } // end Location property
18|
19|     public void TakeATurn(PairOfDice dice) {
20|         Console.WriteLine("{0} is on {1}, ", Name, Location);
21|         dice.Roll();
22|         Console.WriteLine("rolls {0}", dice.Value );
23|         Console.WriteLine(" and lands on " );
24|         for(int i = 0; i < dice.Value; ++i) {
25|             myLocation = Location.Next;
26|         } // end for loop
27|         Console.WriteLine(Location);
28|     } // end TakeATurn method
29| } // end Player class
```

**File: Game.cs**

```
1| public class Game {
```

```
2|     private PairOfDice myDice = new PairOfDice();
3|     private Board myBoard = new Board();
4|     private Player[] players;
5|
6|     public Game(int playerCount) {
7|         players = new Player[playerCount];
8|         for(int i = 0; i < playerCount; ++i) {
9|             players[i] = new Player("Player#" + i, myBoard.StartingLocation);
10|        } // end for loop
11|    } // end Game constructor
12|
13|    public void Play(int rounds) {
14|        for(int i = 0; i < rounds; ++i) {
15|            foreach(Player p in players) {
16|                p.TakeATurn(myDice);
17|            } end foreach loop
18|        } end for loop
19|    } // end Play method
20| } // end Game class
```

### File: MainApp.cs

```
1| public class MainApp {
2|     public static void Main() {
3|         Game g = new Game(2);
4|         g.Play(10);
5|     } // end Main method
6| } // end MainApp class
```

### Build & Execute Instructions

To build these files, do the following:

1. At a command prompt and issue the following command:

```
c:\C#\BoardGame>csc *.cs
```

2. Execute your program with the following command:

```
c:\C#\BoardGame>MainApp
```

3. You will see output resembling the following:

```
C:\C#\BoardGame>mainapp
Player#0 is on BL(Starting Location), rolls 5 and lands on BL(Loc#5)
Player#1 is on BL(Starting Location), rolls 8 and lands on BL(Loc#8)
Player#0 is on BL(Loc#5), rolls 12 and lands on BL(Loc#17)
Player#1 is on BL(Loc#8), rolls 6 and lands on BL(Loc#14)
Player#0 is on BL(Loc#17), rolls 8 and lands on BL(Loc#25)
Player#1 is on BL(Loc#14), rolls 4 and lands on BL(Loc#18)
Player#0 is on BL(Loc#25), rolls 7 and lands on BL(Loc#32)
Player#1 is on BL(Loc#18), rolls 8 and lands on BL(Loc#26)
Player#0 is on BL(Loc#32), rolls 12 and lands on BL(Loc#4)
Player#1 is on BL(Loc#26), rolls 7 and lands on BL(Loc#33)
Player#0 is on BL(Loc#4), rolls 7 and lands on BL(Loc#11)
Player#1 is on BL(Loc#33), rolls 9 and lands on BL(Loc#2)
Player#0 is on BL(Loc#11), rolls 8 and lands on BL(Loc#19)
Player#1 is on BL(Loc#2), rolls 4 and lands on BL(Loc#6)
Player#0 is on BL(Loc#19), rolls 3 and lands on BL(Loc#22)
Player#1 is on BL(Loc#6), rolls 4 and lands on BL(Loc#10)
Player#0 is on BL(Loc#22), rolls 8 and lands on BL(Loc#30)
Player#1 is on BL(Loc#10), rolls 8 and lands on BL(Loc#18)
Player#0 is on BL(Loc#30), rolls 11 and lands on BL(Loc#1)
Player#1 is on BL(Loc#18), rolls 8 and lands on BL(Loc#26)
```

### Code Explained

You will probably notice that the line-by-line explanations are going to be getting smaller as you read on. We are only pointing out new things or things we think are significant.

**File: BoardLocation.cs**

#	Description
3	<code>private BoardLocation myNext;</code> Declare an attribute which is a reference to the same class. That is, each instance of the BoardLocation class holds a reference to an instance of the BoardLocation class. This makes it possible to form a chain of BoardLocation objects. This attribute is represented on Figure 11 as a solid line from the BoardLocation class back to itself.
5	<code>public BoardLocation(string name) {</code> This class has one constructor that takes a single parameter. That means the only way to create a BoardLocation is to provide a name when it is created.
6	<code>myName = name;</code> The lifetime of the parameter, name, is from the { to the } of the constructor. The lifetime of the attribute myName is equal to the lifetime of the object in which it is contained. We store the parameter in an attribute so that the BoardLocation always has a name.
9	<code>public string Name {</code> Declare a property. This is a read-only property because there is only a get section.
13	<code>public BoardLocation Next {</code> Declare a property. This is a read/write property because it has both a get and set section.
15	<code>set { myNext = value; }</code> This is the set section of the property. Notice the use of the variable called “value”. This is a pre-defined variable for setters within a property. Using a setter looks like this: <pre>BoardLocation first = new BoardLocation("First"); BoardLocation second = new BoardLocation("Second"); first.Next = second; second.Next = first;</pre> These two lines use the set section of the Next property. For the first line, the local variable value refers to the object known as second. In the second line, value refers to first.
19	<code>return "BL(" + Name + ")";</code> When we originally wrote this, the output lines were > 80 characters long. We shortened up the output by using BL instead of BoardLocation to make individual output lines shorter. As we work our way through the problem we will be changing output frequently.

**File: Board.cs**

#	Description
2	<code>private BoardLocation myStart;</code> The board maintains a reference to one BoardLocation and calls it myStart. This attribute is represented on Figure 11 as a solid line between the Board class and the BoardLocation class.
5	<code>myStart = new BoardLocation("Starting Location");</code> The Board initializes its myStart attribute. Notice that we are passing a parameter into a constructor. If we did not provide a name, this line would not compile.
7	<code>BoardLocation current = StartingLocation;</code> We are effectively creating a circular linked list. Ultimately we will create a total of 40 BoardLocation objects and they will be connected like Figure 12.
12	<code>current.Next = newNext;</code> We are using the Next property of the BoardLocation class. Notice that we simply perform an assignment; it looks like we are simply accessing an attribute of the class directly. In fact underneath the covers this may or may not cause a function to be called. The underlying system decides how to optimize this. How the system performs this optimization is out of the scope. However, C# does this to encourage us to use properties or methods to access information within an object rather than directly accessing it. This leads to code that is easier to modify. Volume 2 in this series addresses issues surrounding function call overhead.
19	<code>get { return myStart; }</code> We have a read-only property again. In general, expose as little as possible to the outside

#	Description
	world. It is easier to add something than it is to remove it. .NET is able to handle different versions of software running at the same time, even in the same process space. Even so, from a systems engineering perspective, exposing only what is necessary rather than everything is best at the beginning. It reduces the chance of unnecessary dependencies within a system, which allows for change better.

**File: Player.cs**

#	Description
3	<code>private string myName;</code> A Player knows its name. This is an attribute, which means it has the same lifetime as that of the object in which it is contained. It is private, so it is not directly accessible.
4	<code>private BoardLocation myLocation;</code> A Player knows its location. Notice that the player does not know the name of the location or the index of the location, the player knows the location. This is a reference to a BoardLocation object. This attributed is represented in Figure 11 as a solid line with an arrow between Player and BoardLocation.
6	<code>public Player(string name, BoardLocation startingLocation){</code> When constructing a player, you provide both its name and where it starts. The player is in a well defined state after its creation. We do not check for invalid parameters, for example a null BoardLocation. We do not to keep the code short and simple, although we will do so later.
11	<code>public string Name {</code> Define a property. This property is read-only because there is no set section. The Player constructor takes in a name, which we store. The name of the player does not change during the game. There is no need to provide write access.
15	<code>public BoardLocation Location {</code> Define a property. This property is also read-only.
19	<code>public void TakeATurn(PairOfDice dice) {</code> Define a method. Each player takes a turn rolling a pair of dice and moving around the board. The Player is given the dice as a parameter into the TakeATurn() method. Figure 11 shows the relationship between Player and PairOfDice as a dashed line. The Player only uses the PairOfDice during its turn. The lifetime of this relationship is short compared to Player's attributes.
20	<code>Console.Write("{0} is on {1}, ", Name, Location);</code> The Write method does not insert a new line in the output. We produce the output in multiple steps so we can print a before and after snapshot of what has happened during the turn. The {0} and {1} indicate to the Write method where to insert the Name and Location parameters passed in. If you use {0}, you must provide one additional parameter. If you use {3}, you must provide 4 additional parameters. If you do not, the Write() method will throw an exception, System.FormatException, and terminate your program. (This is not strictly true, it will terminate the current thread. The fact that your program only has one thread means, in effect, your whole program terminates.)
24	<code>for(int i = 0; i &lt; dice.Value; ++i) {</code> We move forward a number of locations equal to the current value of the dice object. Notice that we are accessing a property. If this PairOfDice object were used in other parts of the program at the same time, e.g. a multi-threaded application, this would not be thread-safe. This is a simple program and we are not explicitly using multi-threading.
25	<code>myLocation = Location.Next;</code> Set my attribute equal to the next neighbor of my current location. We are moving through the links of the BoardLocation objects. The BoardLocation objects form a circular linked list of objects. Each knows its next neighbor.

**File: Game.cs**

#	Description
2	<code>private PairOfDice myDice = new PairOfDice();</code> One of the three attributes within a Game is the PairOfDice. Declare the attribute initialize it. The system executes this initialization once for each Game object created. The initialization could have been in the Constructor, it is here to demonstrate that simple initialization can go here instead of in the constructor. Figure 11 shows this attribute as a solid line between Game and PairOfDice.
3	<code>private Board myBoard = new Board();</code> Figure 11 shows this attribute as a solid line between Game and Board.
4	<code>private Player[] players;</code> Declare an array. We do not initialize it here because we will not know its size until we get to the constructor. Figure 11 shows this attribute as a solid line between Game and Player.
6	<code>public Game(int playerCount) {</code> Define a constructor that takes in the number of players for a game.
7	<code>players = new Player[playerCount];</code> Create an array large enough to hold the number of players requested when calling the constructor. We still have not created any players. First we declared an array. Later, we allocate or defined the array. After that we fill it in with objects. Often the first two steps are together on one line. We could not do this for two reasons. First, we do not know until the constructor is called how many players we need to create. Second, the array should be defined as an attribute since we want the player array to have the same lifetime as the game. If we try and declare the array within the {} of the constructor, the array's lifetime will be that of the constructor instead of the lifetime of the object.
9	<code>players[i] = new Player("Player#" + i, myBoard.StartingLocation);</code> Create a player. For its name, concatenate the string "Player#" and the loop variable. The second parameter we pass in is where we want the player to begin. The Board created all of the BoardLocation objects and knows the starting location. We get the starting location from the Board and pass it into the Player constructor. After creating a player, the resulting reference is recorded in the array.
13	<code>public void Play(int rounds) {</code> Define a public method. This is where the game is finally played. Whoever calls the method provides a number of rounds to play.
14	<code>for(int i = 0; i &lt; rounds; ++i) {</code> Outer loop controlling the number of rounds to play.
15	<code>foreach(Player p in players) {</code> Inner loop making sure we give each player a turn. If we play 10 rounds with 2 players, line 18 will execute 20 times.
16	<code>p.TakeATurn(myDice);</code> Tell a player to take a turn. The player needs to use the PairOfDice object so the Game passes in the reference to the dice. This allows the player to send Roll() to the dice to know how far it should move.

**File: MainApp.cs**

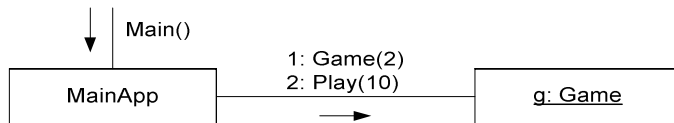
#	Description
3	<code>Game g = new Game(2);</code> Create a Game object and ask it to have 2 players.
4	<code>g.Play(10);</code> Ask the game to play 10 rounds. During each of the 10 rounds, each player will take one turn.



## Object Collaboration

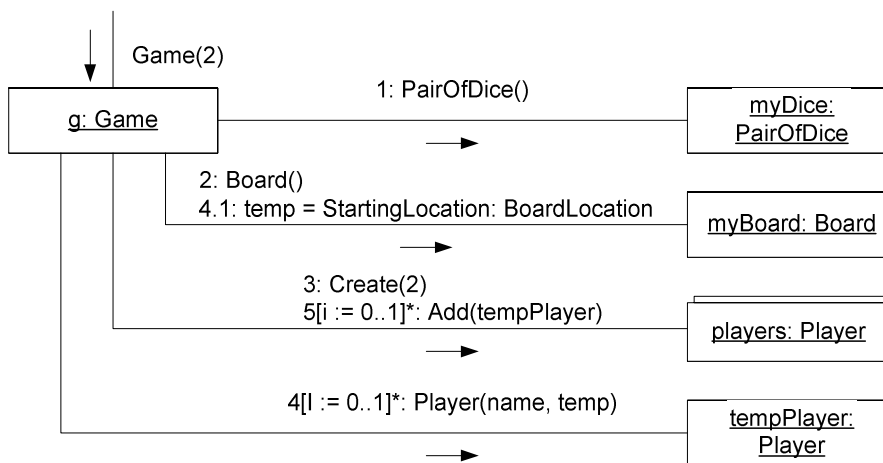
The collaboration between all of the objects is too complex to put into one diagram. Instead, we present several separate diagrams.

The MainApp class creates a game and tells it to play:



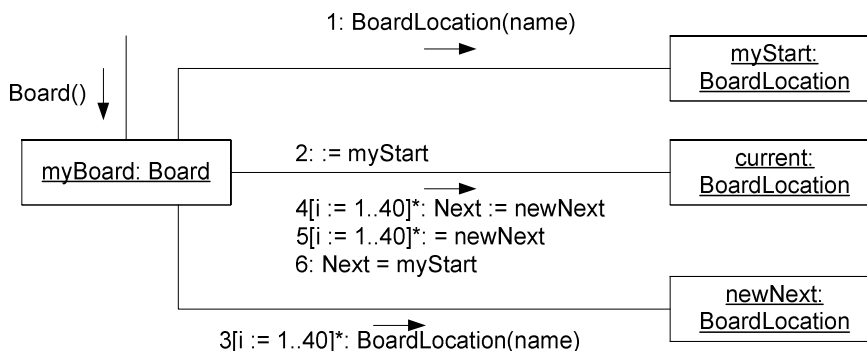
**Figure 13: UML – Collaboration: MainApp → Game**

Object Creation for the game begins with creating the dice, then the board then the players. To create the players, the Game first creates an array big enough to hold the Player objects. It then creates the players and adds them to the array in a loop.



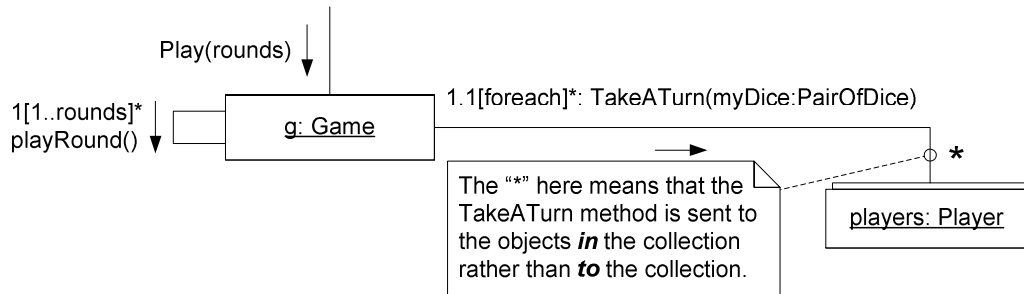
**Figure 14: UML – Collaboration: Game.Create(int)**

Board Creation is a bit more complex. After creating its starting location, the board uses a few temporary variables; one to hold the current, one to hold the newly created BoardLocation, called newNext. The Board assigns current to myStart, it then repeatedly creates a new BoardLocation, holding its reference in newNext, makes newNext the Next of the current BoardLocation, then sets current equal to newNext.

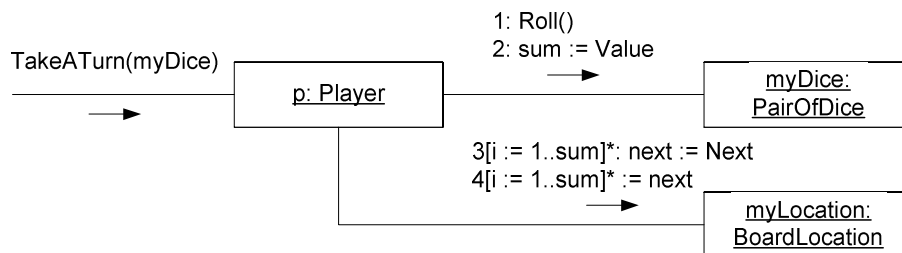


**Figure 15: UML – Collaboration: Board.Create()**

After the Game has created all of its objects, MainApp next tells the Game to Play. The game iterates once for each round. For each round, the Game tells each player to TakeATurn:

**Figure 16: UML – Collaboration: Game.Play(int)**

Finally, each `Player` takes a turn. They do this by first sending `Roll` to the `PairOfDice`. They then move one `BoardLocation` at a time using the `Next` property and finally record where they landed in their `myLocation` attribute. Note that we are not showing any of the output messages:

**Figure 17: UML – Collaboration: Player.TakeATurn(PairOfDice)**

## Review

- An object can hold references to any other object, even other objects of the same class. A `BoardLocation` points to another instance of the `BoardLocation` class.
- Constructors can take parameters.
- If a class has a constructor with parameters, there is no zero argument constructor unless we specifically write one. Note that we can write any number of constructors in our classes.
- Properties can be read-only or read-write.
- The `Write` and `WriteLine` methods of the `Console` class allow parameters to be inserted using `{0}`, `{1}`, etc. An exception might get thrown if you do not pass enough parameters.
- There are 4 steps to using arrays.
  - Declare an array: `Player[] anArray;`
  - Allocate memory for the array: `anArray = new Player[2];`
  - Put something in the array: `anArray[0] = new Player(...);`
  - Use the array: `foreach(Player p in anArray) { Console.WriteLine(p.Name); }`
- Arrays are 0-based. When you allocate an array of size 10, the valid indexes are 0 through 9. This might be written as `[0..10)`. Trying to work outside of this range will throw an exception.
- The `foreach` keyword allows iteration through the contents of an array. It does more than this but we have only used it on arrays.
- Attributes can be initialized using code that appears to be outside of the constructor. See `Game.cs` for examples of this. In reality, the code is inserted into the constructor for you.

- The lifetime of a variable is determined by the scope in which it is defined. A variable in a method lives from the line of code where it is declared to the end of the method. Variables defined in a while loop live only inside the while loop. Attributes logically follow this rule as well. They are defined within the {} of a class so they live as long as their containing object.

- One notable exception is the lifetime of loop variable defined in the loop. For example:

```
void method() {  
    for(int i = 0; i < 10; ++i) {  
    }  
    // i still exists outside of the loop  
}
```

Microsoft Visual C++ does the same thing, which is in direct conflict with the C++ standard. This is a deficiency in both Visual C++ as well as C#.

## Iteration 1a: No New Functionality

This section covers a few background topics that will improve our code without adding any functionality. How is this possible? We will do the following:

- Allow our classes to co-exist with other classes, even classes with the same names.
- Improve the reliability of our classes by checking that we are not provided with invalid parameters.
- Processing command line parameters.

Think of this as a quick sidebar where we only make minor changes. However, the remainder of the examples will include the subjects we cover here.

### Namespaces

#### A Problem

We have written seven classes so far. In addition, the .NET framework has many classes. How do we organize classes to avoid name conflicts? The .NET solution is to put related classes into a namespace. Two classes with the same name but different namespaces can co-exist in the same program.

Review [5] for a better understanding of how decide what constitutes a good grouping of classes. Read the remainder of this section to see how to use namespaces in this project.

#### A Solution

Here is the MainApp.cs class updated to use namespaces:

```
1| namespace BoardGame {
2|     public class MainApp {
3|         public static void Main() {
4|             Game g = new Game(2);
5|             g.Play(10);
6|         } // end Main method
7|     } // end MainApp class
8| } // end BoardGame namespace
```

To introduce a namespace, use the keyword `namespace` followed by `{`. Everything up to the matching `}` is within the namespace. Any new names introduced now include the name of the namespace as part of their fully qualified name. In this example, `MainApp` is now really `BoardGame.MainApp`. A namespace can span multiple files. This means we can simply follow the above example for each of our source files, recompile and execute.

In this example we do not tab everything within the namespace to visually show the containment relationship between a namespace and its classes. C# does not care. We avoid the tab because adding a tab for every line within namespace means that nearly every line of every file will have a tab at the beginning since we choose to put our classes in namespaces. Here is an example of what the above file would look like indenting everything within the namespace:

```
1|     namespace BoardGame {
2|         public class MainApp {
3|             public static void Main() {
4|                 Game g = new Game(2);
5|                 g.Play(10);
6|             } // end Main method
7|         } // end MainApp class
8|     } // end BoardGame namespace
```

Your standard may vary. Whatever it is, follow it.

## Build & Execute Instructions

To update all of your classes to use namespaces, follow these instructions:

1. Edit each file, adding “namespace BoardGame {” as the first line.
2. Add a matching closing } to each file as the last line.
3. Compile and generate an executable:

```
c:\C#\BoardGame>csc *.cs
```

4. Execute your program with the following command:

```
c:\C#\BoardGame>MainApp
```

## Parameter Checking

### A Problem

The Game creates several Player objects. It does with the following code:

```
players[i] = new Player("Player#" + i, myBoard.StartingLocation);
```

In this example, we provide two parameters to the constructor, a name and a BoardLocation object. Right now nothing prevents passing in null for either the name or the starting location. If a Player’s name is null, the output might be hard to read but the program will still execute. On the other hand, if the Player’s starting location is null, the program will not operate. Even worse, the problem will not occur until a player takes a turn, which is well after our code caused the problem. It might be difficult to track down where the problem actually is versus where we are able to observe it.

### A Solution

We can add some level of checking into our code to check that input parameters are valid in some way. For now we will perform two kinds of checks:

- Parameters that are null, which should not be null.
- Parameters that have values outside some range.

If our code encounters either of these problems, we will simply throw an exception back to the calling code and let that code deal with the problem.

An exception is a standard error-handling mechanism. C# has four keywords for exception handling: throw, try, catch, and finally. The throw keyword generates an exception. When we throw an exception, the system starts looking for a matching try/catch block. Right now our goal is simple, stop the program if we encounter any serious errors. We can do this by simply throwing exceptions. We will not address more detailed exception handling until later.

Since we are not writing any code to handle these exceptions, then the system will terminate. This may sound a bit extreme; however the code would have failed anyway. We are just making it happen closer to where the code that caused the problem. By checking sooner we are closing a feedback loop sooner and making the problem easier to solve.

### Exceptions

First we will take a look at what we will do to check and, if necessary, throw an exception. Then we will discuss exception handling. This is our first look into using exceptions. We will see much more quite a bit later.

If we were to fix player, first we need some policy to help determine how we might manage null parameters. Let’s use a simple one; we do not accept null parameters. There might be cases where a null parameter is OK. If this is the case, we will document our method so that someone using it knows that null is OK.

Given the simple policy, let's protect the Player's constructor so that null parameters will fail sooner rather than later:

### Example: Player.Player

```

1| public Player(string name, BoardLocation startingLocation) {
2|     if(name == null) {
3|         throw new System.ArgumentNullException("name");
4|     } // end if
5|     if(startingLocation == null) {
6|         throw new System.ArgumentNullException("startingLocation");
7|     } // end if
8|     myName = name;
9|     myLocation = startingLocation;
10| } // end Player constructor

```

#	Description
2	<b><code>if(name == null) {</code></b> Compare the parameter to null.
3	<b><code>throw new ArgumentNullException("name");</code></b> The name parameter is null, create an exception, <code>ArgumentNullException</code> , and throw it back to the calling code. We pass in "name" to indicate that the parameter named "name" was null. Our method immediately terminates and returns back to the calling method. If the calling method does not specifically handle this code, then that method will also immediately terminate and return back to that calling method. This continues until either some code handles the exception or the program terminates.
5-6	Same comments as above. The only difference is the name of the parameter we provide to the constructor of the <code>ArgumentNullException</code> .

## Build & Execute Instructions

To convince yourself that this does in fact do something, you can write the following code:

```

1| namespace BoardGame {
2|     public class MainApp {
3|         public static void Main() {
4|             new Player(null, null);
5|             Game g = new Game(2);
6|             g.Play(10);
7|         } // end Main method
8|     } // end MainApp class
9| } // end BoardGame namespace

```

The first line of code in `Main()` attempts to create an instance of `Player` by passing in null for both parameters. Compile and execute the system. The .NET execution environment will display a just in time debugging dialog asking you if you want to debug the program. Select no and you will get the following output:

```
c:\C#\BoardGame>mainapp
```

```

Unhandled Exception: System.ArgumentNullException: Value cannot be null.
Parameter name: name
   at BoardGame.Player..ctor(String name, BoardLocation startingLocation)
   at BoardGame.MainApp.Main()

```

This handles null arguments, what about values out of range? The .NET framework also has an exception class called `ArgumentOutOfRangeException`. Here's an example from `Game.cs`:

```

1| using system;
2| public Game(int playerCount) {
3|     if(playerCount < 1 || playerCount > 4) {
4|         throw new ArgumentOutOfRangeException(
5|             "playerCount", playerCount, "should be 2..4"
6|         );
7|     } // end if
8|     players = new Player[playerCount];

```

```

9|         for(int i = 0; i < playerCount; ++i) {
10|             players[i] = new Player("Player#" + i, myBoard.StartingLocation);
11|         } // end for loop
12|     } // end Game constructor

```

In this example, it does not make sense to create a game with less than one player. It is possible to create an array of size 0. It is not too interesting but nothing will break. None of our code assumes the array is any size. An array of size 0 means when it is time to tell each player to TakeATurn(), there will be no players who actually do anything. The program will not do much, but it will not break. Later on we will probably change this to require 2 or more players, for now 1 will suffice.

To show this in action, we change MainApp.cs again:

```

1| namespace BoardGame {
2|     public class MainApp {
3|         public static void Main() {
4|             Game g = new Game(-1);
5|             g.Play(10);
6|         } // end Main method
7|     } // end MainApp class
8| } // end BoardGame namespace

```

Execute this program, and as before a dialog will popup. After closing the dialog, you will see a message like the following:

```
c:\C#\BoardGame>mainapp
```

```

Unhandled Exception: System.ArgumentOutOfRangeException: should be 2..4
Parameter name: playerCount
Actual value was -1.
   at BoardGame.Game..ctor(Int32 playerCount)
   at BoardGame.MainApp.Main(String[] args)

```

The system prints out the string we provide in the constructor of the ArgumentOutOfRangeException.

## File: Assert.cs

Before we actually update all of our code to support parameter checking, we will first introduce a simple utility class that will make it much easier both to write and to read such checks:

```

1| using System;
2| namespace BoardGame {
3|     public sealed class Assert {
4|         public static void NotNull(Object obj, String paramName) {
5|             if(obj == null){
6|                 throw new ArgumentNullException(paramName);
7|             } // end if
8|         } // end NotNull method
9|
10|         public static void Argument
11|             (bool b, String paramName, Object actualValue, String message) {
12|             if(!b){
13|                 throw new ArgumentOutOfRangeException
14|                     (paramName, actualValue, message);
15|             } // end if
16|         } // end Argument method
17|     } // end Assert class
18| } // end BoardGame namespace

```

#	Description
2	<p><b>public sealed class Assert {</b></p> <p>Define a new class called Assert. It is sealed, meaning no other classes can subclass off of this class. While we cover inheritance in iteration 2, for now you can safely ignore this keyword.</p> <p>This class does what we were already doing for null and range checks. Rather than writing:</p> <pre> if(name == null) {     throw new System.ArgumentNullException("name"); } </pre> <p>We can now write:</p>

#	Description
	<code>Assert.NotNull(name, "name");</code>
4	<p><b><code>public static void NotNull(Object obj, String paramName) {</code></b></p> <p>This is the method we will use to check for null. Provide the object to be checked along with the name of the parameter. Rather than writing:</p> <pre>if(name == null) {     throw new System.ArgumentNullException("name"); }</pre> <p>We can now write:</p> <pre>Assert.NotNull(name, "name");</pre> <p>We have already seen the static keyword. It means we can use this method without an instance of the class. As you can see above, we use the name of the class followed by the method name rather than creating an instance of the Assert class and using it.</p>
10	<p>This method replaces checking for parameters within a range. Rather than writing:</p> <pre>if(playerCount &lt; 1) {     throw new ArgumentOutOfRangeException(         "playerCount", playerCount, "should be &gt; 1"     ); }</pre> <p>We can now write:</p> <pre>Assert.Argument(playerCount &gt; 1, "playerCount", playerCount, "should be 2..4");</pre>

Now that we can easily add parameter checking, a question to ask is where should we do so? Initially we had no checking at all. On the other side of the spectrum is checking everything everywhere. We can go to the other extreme but we will not. Generally once we have found such a range of extremes we will pick somewhere in the middle. We will check all constructors to make sure objects have valid parameters with which to start. We will also check the methods in Game.

We will check constructors since that is the place where we initialize an object. In most cases an object will be well defined after the constructor. If there are problems with an object it will often occur during construction. Adding checks on constructor parameters will catch quite a bit.

We check the game for two reasons. First, it is the object that represents the entire system. We will often call this a façade. A façade presents a face to the world for an entire system or subsystem. Checking the parameters on a façade is generally a very good idea because a lot of problems will occur here as well. A second reason, which relates to the first, is in the next section we will be taking in command line parameters. These parameters are provided on the command line and they might not be valid values. By putting checks in the Game class we will capture invalid parameters.

Give these two kinds of checks, the following methods are candidates for adding parameter checking:

- `Player.Player(string name, BoardLocation startingLocation)`  
`Assert.NotNull(name, "name");`  
`Assert.NotNull(startingLocation, "startingLocation");`
- `BoardLocation.BoardLocation(string name)`  
`Assert.NotNull(name, "name");`
- `Game.Game(int playerCount)`  
`Assert.Argument(playerCount > 1, "playerCount", playerCount, "should be > 1");`
- `Game.Play(int rounds)`  
`Assert.Argument(rounds > 0, "rounds", rounds, "should be > 0");`



## Command Line Parameters

### A Problem

Our system currently plays with 2 players and 10 rounds. Currently, the only way to change that is to change the source code, recompile and re-run. A common way to change this kind of behavior in a system is to read parameters from the command line.

The final addition to our system for iteration 1a is to use command line parameters to optionally specify the number of players and the number of rounds. We will update MainApp.cs to have a default of 2 Players and to play 10 rounds. We are going for ultra simple, so if there is only one command line parameter, it specifies the number of players. If there are two, the first is the number of players; the second is the number of rounds. There will be no way to specify the number of rounds without specifying the number of players. If there are more than two parameters, the extra parameters are ignored. Finally, if either of the parameters cannot be converted to numbers, e.g. you type aaa, the system will terminate and throw System.FormatException.

### A Solution

#### File: MainApp.cs

```
1| using System;
2| namespace BoardGame {
3|     public class MainApp {
4|         public static void Main(string[] args) {
5|             int playerCount = 2;
6|             int roundCount = 10;
7|
8|             if(args.Length > 1) { roundCount = Int32.Parse(args[1]); }
9|             if(args.Length > 0) { playerCount = Int32.Parse(args[0]); }
10|
11|             Game g = new Game(playerCount);
12|             g.Play(roundCount);
13|         } // end Main method
14|     } // end MainApp class
15| } // end BoardGame namespace
```

#	Description
4	<b><code>public static void Main(string[] args) {</code></b> The signature for Main changes if we want to handle command line arguments. We define a single parameter, an array of Strings. There can be only one Main method. It either takes no parameters or it takes one, an array of string.
5	Set a default number of players.
6	Set a default number of rounds.
8	Are there at least two parameters? If so, use the second one as the round count. Take the value and pass it to the Parse method of the Int32 class. This method will convert a string to an int, or throw a System.FormatException if the command line parameter cannot be converted to an int.
9	Is there at least one parameter? If so, use it to reset the player count.
11	Create the game with the provided number of players. Note that since we've added parameter checking, if the player count is < 1, the Game constructor will throw an exception.
12	Play the number of rounds specified. Here again, if the round count is < 1 the Play method will throw an exception because we are checking parameters.

### Build & Execute Instructions

Now we have three ways to start out application:

1. c:\C#\BoardGame>MainApp
2. c:\C#\BoardGame>MainApp 4

```
3. c:\C#\BoardGame>MianApp 3 18
```

The first example runs with 2 players and 10 rounds. The second plays with 4 players and 10 rounds. The third plays with 3 players and 18 rounds.

## Iteration 2: Inheritance & Polymorphism

In this section we will extend our system. We will use inheritance and polymorphism to solve the problem. We will begin with new requirements. We will continue with a picture of the classes in the new system. We will summarize the changes to the code and then cover those changes. We will finish with a review.

### New Requirements

First we will set the board size. We did not specifically say how many locations existed on the board so we will define it as 40. Using a pair of dice, a player's roll will average 7. On average a player will roll the pair of dice 6 times before covering the distance around the board.

As players move around the board, different things happen when players either pass over or land on various board locations. For example, later on players will be able to purchase some of the board locations. To be able to do this, players will need money and a reliable source of income. When a player completes a loop around the board, they earn a salary of 200. They earn this salary during any turn where they either land on or pass over the original starting location.

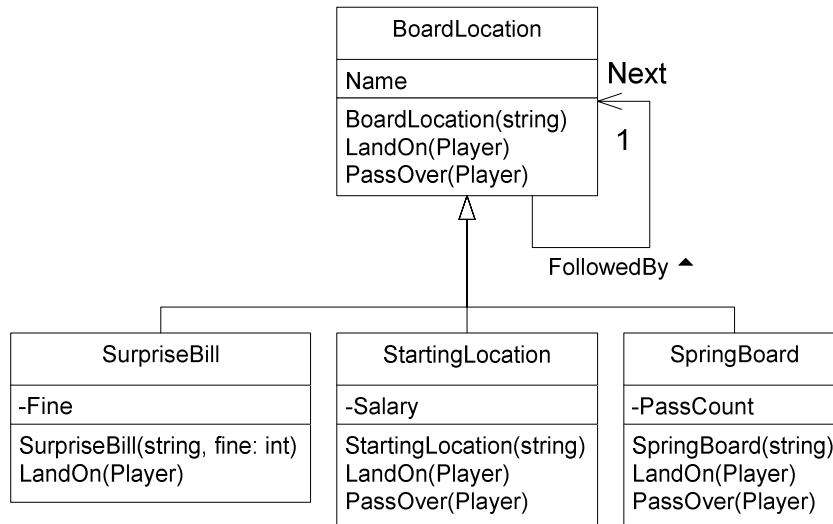
As with life, some times things need repairing and that costs money. There are some board locations that, when a player lands on them, cost the player some amount. We might call this a surprise bill. When a player lands on such a location, the location charges them some amount. For now we will only have two of these locations, one will charge a fine of 100, the other a fine of 75.

There is one particular location known as the SpringBoard. The SpringBoard causes a player to jump directly ahead a certain number of locations. By directly, this means the player does not pass the other squares. The distance a player is sprung forward depends on the number of times any player has passed over the SpringBoard without landing on it. The distance is equal to 3 times the number of times a player has passed the SpringBoard in the past, up to a maximum of 39 locations. If no players have passed the SpringBoard, the player does not spring forward and the count does not increase. If the SpringBoard has been passed 4 times by any of the players, it will spring the landing player ahead 12 Locations. Finally, once the SpringBoard has sprung, it resets its count back to 0. If a player happens to spring pass the starting location, they do not receive their salary,

We have four specific locations on the board where something happens. This table summarizes those locations:

#	Name	Description
0	StartingLocation	When a Player passes over or lands on this location, she receives her salary of 200.
2	SurpriseBill	When a player lands on this location, they pay a fine of 100. Nothing happens if the player passes over this location.
19	SpringBoard	When a player lands on this location they are sprung forward a number of locations. If a player merely passes over the location, the spring is wound a little more tightly, up to a maximum count of 13 passes.
26	SurpriseBill	When a player lands on this location, they pay a fine of 75. Nothing happens if the player passes over this location.

We can represent the static structure of these new classes in a hierarchy:



**Figure 18: UML – Structure: Board Location Hierarchy for Iteration 2**

## Background: Inheritance & Polymorphism

We have 40 locations around the board that have similar characteristics:

- They have a name and they know their following neighbor.
- They represent a place where a player resides between turns.
- Things might happen when players land on some locations.
- Things might happen when players pass over some locations.

Our current **BoardLocation** class handles the first two list items. It does not handle the last two. If we can somehow make it support those last two items, then it will serve as a good base class from which we can inherit the new special locations.

Inheritance allows the definition of one class to serve as a basis of definition for other classes. In this case we have an opportunity to use inheritance for all of the **BoardLocation** classes because there are several common characteristics shared between them.

Polymorphism allows objects that share some common characteristics to respond to the same request in different ways. One of the most common examples is that different shapes know how to draw themselves. Ask a square to draw itself and you expect to see something resembling a square. Ask a circle to draw itself and you expect a circle to appear.

A key idea about polymorphism is who makes the decision. In the shape example, there is one request, **Draw()**, that causes different things to happen depending on which object receives the request. The receiver decides what happens, not the sender. The sender decides what to ask for, the receiver decides what to do.

## Method and message

We have used two different terms without really defining them: **Method** & **Message**. **Method** is the code you write and that the system executes. A method looks just like a function or subroutine, although there is one key difference. A method operates within an additional level of scope that functions do not have. That additional scope is the current object (or class for static methods). Otherwise we can think of methods and functions as the same thing.

A message is a request to do some work. A user of an object sends a message to that object. The object executes a method. This might sound no different than a method but there is a key difference. In the shape example, we ask a shape to draw itself. That is the request or message. The shape does

what it is asked to do, it draws itself. That is the method. However, assuming we do not know whether we are asking a square or circle to draw itself, we do not know what will happen until we ask.

How does this apply to our current problem? In our case we have two different requests:

- A request to land on a location.
- A request to pass over a location.

What happens in both of these situations depends on which location receives the message. Sending the land on message to a StartingLocation results in the player receiving 200. Landing on the SpringBoard results in the player moving forward some number of locations based on what has happened to the location in the past.

The same can be said of passing over a location. When a player passes over StartingLocation, they receive their salary. When a player passes over SpringBoard, the SpringBoard increases its spring counter.

## Build 1: A Quick Start

Rather than continue the discussion, we will get one of the locations working. We will then summarize what it takes to get inheritance and polymorphism to work in C# and continue with the remainder of the locations.

Note that for each of the following files we have removed most of the code that is unchanged from the previous version.

### Updates to: BoardLocation.cs

```
1| // Add the following two methods
2| public virtual void LandOn(Player currentPlayer) { }
3| public virtual void PassOver(Player currentPlayer) { }
4|
5| // Update the following method
6| public override string ToString() {
7|     return GetType().Name + "(" + Name + ")";
8| }
```

### File: StartingLocation.cs

```
1| using System; // we use Console, which is in the System namespace
2| namespace BoardGame {
3|     public class StartingLocation : BoardLocation {
4|         private const int SALARY = 200;
5|
6|         public StartingLocation(string name) : base(name) {
7|             // nothing to do in this constructor
8|         } // end StartingLocation constructor
9|
10|        public override void LandOn(Player p) {
11|            Console.WriteLine("\t{0} lands on {1} and receives {2}",
12|                p.Name, Name, SALARY);
13|            p.Cash += SALARY; // update the Player's balance
14|        } // end LandOn method
15|
16|        public override void PassOver(Player p) {
17|            Console.WriteLine("\t{0} passed over {1} and receives {2}",
18|                p.Name, Name, SALARY);
19|            p.Cash += SALARY; // add SALARY to the Player's balance
20|        } // end PassOver method
21|    } // end StartingLocation class
22| } // end BoardGame namespace
```

### Updates to: Board.cs

```
1| // Update the Board constructor
2| public Board() {
```

```
3|     myStart = new StartingLocation("Start");
4|     // remainder of the code is unchanged
5| } // end Board constructor
```

### Updates to: Player.cs

```
1| // Add an attribute to Player
2| private int myCash;
3|
4| // Update the Player constructor
5| public Player(string name, BoardLocation startLoc, int startCash) {
6|     Assert.NotNull(name, "name");
7|     Assert.NotNull(startingLocation, "startingLocation");
8|     Assert.Argument(startCash > 0, "startCash", startCash, "not > 0");
9|
10|     myName = name;
11|     myLocation = startLoc;
12|     myCash = startCash;
13| } // end Player constructor
14|
15| // Add a Cash property
16| public int Cash {
17|     get { return myCash; }
18|     set { myCash = value; }
19| } // end Cash property
20|
21| // Update Location property to have a set section
22| public BoardLocation Location {
23|     get { return myLocation; }
24|     set { myLocation = value; }
25| } // end Location property
26|
27| // Update Player.TakeATurn
28| public void TakeATurn(PairOfDice dice) {
29|     Console.WriteLine("{0} on {1} has {2} ", Name, Location, Cash);
30|     dice.Roll();
31|     Console.WriteLine("\trolls {0} ", dice.Value );
32|     Location = Location.Next;
33|     for(int i = 1; i < dice.Value; ++i) {
34|         Location.PassOver(this);
35|         Location = Location.Next;
36|     } // end for loop
37|     Console.WriteLine("\twill land on {0}", Location);
38|     Location.LandOn(this);
39|     Console.WriteLine("\tand now has {0}", Cash);
40| } // end TakeATurn method
```

### Updates to: Game.cs

```
1| // Add a constant to the Game
2| private const int STARTING_CASH = 1500;
3|
4| // Update constructor to pass in cash when creating Player objects
5| public Game(int playerCount) {
6|     // remainder of constructor is the same
7|     for(int i = 0; i < playerCount; ++i) {
8|         players[i] = new Player(
9|             "Player#" + i, myBoard.StartingLocation, STARTING_CASH
10|         ); // end of line which creates Player object
11|     } // end for loop
12| } end Game constructor
```

### Build & Execute Instructions

Follow these instructions to see the change to your system:

1. Edit the BoardLocation.cs file. Add the two virtual methods.
2. Create StartingLocation.cs.

3. Compile to make sure these changes work.

```
c:\C#\BoardGame>csc *.cs
```

4. Update Board.cs. Make sure to initialize myStart to an instance of StartingLocation.

5. Update Player.cs.

- Add a cash attribute.
- Update the constructor to take cash as a third parameter and make sure to set the cash attribute in the body of the constructor.
- Add a set section to the Location property.
- Rewrite TakeATurn. Note that besides calling the LandOn and PassOver methods, it also updates the output. This will make it easier to see that different things are happening when landing on or passing over some of the BoardLocation objects.

6. Change Game.cs to pass in the cash parameter to the Player when constructing it.

7. Compile and execute your program. Note this is only a partial snapshot of the output.

```
c:\C#\BoardGame>MainApp
Player#0 on StartingLocation(Start) has 1500
    rolls 4
    will land on BoardLocation(Loc#4)
    and now has 1500
Player#1 on StartingLocation(Start) has 1500
    rolls 6
    will land on BoardLocation(Loc#6)
    and now has 1500
Player#0 on BoardLocation(Loc#4) has 1500
    rolls 9
    will land on BoardLocation(Loc#13)
    and now has 1500
Player#1 on BoardLocation(Loc#6) has 1500
    rolls 5
    will land on BoardLocation(Loc#11)
    and now has 1500
Player#0 on BoardLocation(Loc#13) has 1500
    rolls 6
    will land on BoardLocation(Loc#19)
    and now has 1500
Player#1 on BoardLocation(Loc#11) has 1500
    rolls 7
    will land on BoardLocation(Loc#18)
    and now has 1500
Player#0 on BoardLocation(Loc#19) has 1500
    rolls 6
    will land on BoardLocation(Loc#25)
    and now has 1500
Player#1 on BoardLocation(Loc#18) has 1500
    rolls 5
    will land on BoardLocation(Loc#23)
    and now has 1500
Player#0 on BoardLocation(Loc#25) has 1500
    rolls 2
    will land on BoardLocation(Loc#27)
    and now has 1500
Player#1 on BoardLocation(Loc#23) has 1500
    rolls 6
    will land on BoardLocation(Loc#29)
    and now has 1500
Player#0 on BoardLocation(Loc#27) has 1500
    rolls 8
    will land on BoardLocation(Loc#35)
    and now has 1500
Player#1 on BoardLocation(Loc#29) has 1500
    rolls 5
```

```

    will land on BoardLocation(Loc#34)
    and now has 1500
Player#0 on BoardLocation(Loc#35) has 1500
    rolls 3
    will land on BoardLocation(Loc#38)
    and now has 1500
Player#1 on BoardLocation(Loc#34) has 1500
    rolls 11
    Player#1 passed over Start and receives 200
    will land on BoardLocation(Loc#5)
    and now has 1700

```

## Code Explained

### Update: BoardLocation.cs

#	Description
	<p>Step 1 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>We have selected BoardLocation as the base class for a hierarchy of BoardLocation classes.</p>
2	<pre><b>public virtual void LandOn(Player currentPlayer) {}</b></pre> <p>Steps 2 &amp; 3 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>This is the first line that makes this class look like it has been designed to be a base class.</p> <p>Define a method called LandOn(). This method is virtual, meaning that subclasses can override the version provided in the BoardLocation with their own special behavior. The method body is empty, which means that the default behavior is to do nothing. The method body being empty is a side effect of approaching the problem in stages, or iteratively.</p> <p>This method serves as a “hook” for subclasses. This is a point of flexibility in the system where subclasses can modify behavior. The overall behavior of the BoardLocation class is open to extension by allowing subclasses to replace the behavior defined in the base class. It is closed in that only these two methods may be replaced by subclasses for polymorphic behavior.</p> <p>See the open closed principle in [6].</p>
3	<pre><b>public virtual void PassOver(Player currentPlayer) {}</b></pre> <p>Step 2 &amp; 3 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>This is the second “hook” method for subclasses. The two events in this system that seem to make a difference are passing over a location and landing on a location.</p>
7	<pre><b>return GetType().Name + "(" + Name + ")";</b></pre> <p>The previous version hard-coded the name to be “BL”. BL represented an abbreviated class name. Now that we have a hierarchy of classes, hard-coding the name is no longer appropriate. This version prints the name of the class of the current object.</p> <p>How does this work? See “Appendix: Virtual Methods Explained” starting on page 129.</p>

### File: StartingLocation.cs

#	Description
1	This class performs output using Console.WriteLine. Include this line to avoid writing System.Console.WriteLine(...).
2	This class belongs in the BoardGame namespace along with all the other classes in this system.
3	<pre><b>public class StartingLocation : BoardLocation {</b></pre> <p>Steps 4 &amp; 5 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>This new class, StartingLocation, inherits from BoardLocation. Everything in the base class is also in the derived class (by default). All the attributes, properties and methods in the base class are also in the derived class (so far).</p> <p>Inheritance is a strong form of coupling between classes. A change in the base class might violate the derived class, or the two classes may diverge over time. Inheritance is a subtle tool</p>



#	Description
	to be used sparingly. Note that this syntax is very close to C++. The notable exception is that you do not specify “public”, “private” or “protected”. In C# there is only what C++ would have called public inheritance.
4	Define a constant, the amount of money that a Player receives when passing over or landing on a location.
5	Notice that we do not add definitions for the myName and myNext attributes. This class gets those by virtue of the fact that it is a subclass. A very common error is to redefine the attributes or methods already inherited from the base class. We did not do that here.
6	<pre><b>public StartingLocation(string name) : base(name) {}</b></pre> <p>A requirement for Step 5 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>This is a constructor. The commentary on line 3 regarding inheritance is a bit of a lie to adults. Constructors are quasi-inherited. A subclass can access base class constructors from the subclass constructors. We show this above with the expression “: base(name)”.</p> <p>Base class constructors are not part of the interface of a derived class, like other inherited methods. This means that if we do not add the constructor, then StartingLocation would not have a constructor that takes in the name, even though the base class happens to provide one.</p> <p>Furthermore, the base class has a constructor that requires a parameter be passed to it. When we try to construct a BoardLocation and do not pass in any parameters, the compiler complains that there is no suitable constructor. This means that we must add a constructor to the StartingLocation class or it will not compile.</p> <p>When we construct an instance of a derived class, we also construct an instance of a base class. The derived class’ constructor happens after the base class constructor. Objects are created from top to bottom. When they go away, they are removed bottom to top.</p> <p>The way we pass parameters to a base class constructor is to use a “:” followed by the word base and the parameters. The StartingLocation class takes one parameter, its name. It passes this value up to the base class before executing any of its initialization code. The base class records the name. After the base class constructor is finished, execution continues with the derived class constructor. Finally, construction of the StartingLocation object is done.</p>
10	<pre><b>public override void LandOn(Player currentPlayer) {</b></pre> <p>Step 6 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>In some cases we want to take what the base class has to offer. In this case we want to explicitly replace what the base class does, nothing, with what we need to do. We match the signature of the method in the base class and replace virtual with override. (If we were to create a subclass of StartingLocation, then we would just keep the override in place.)</p> <p>C# requires us to specify which methods should allow for polymorphism by using the keyword virtual. This is because virtual methods do cause a slight overhead to invoke. (A well designed system using inheritance and polymorphism appropriately will perform better than a system using the most direct equivalent, conditional logic or switch statements.)</p> <p>C# also requires us to explicitly state when we are replacing behavior inherited from a base class by using the keyword override. This is so our code better states our intentions. If things change over time, the compiler is more likely to be able to catch it. It is also necessary to support some more advanced uses of inheritance.</p> <p>When we send a message to a BoardLocation, any board location that defines its own LandOn or PassOver method will have its method executed instead of the one in the base class.</p>
13	<pre><b>currentPlayer.Cash += SALARY;</b></pre> <p>The effect of landing on StartingLocation is to receive a salary. This line does just that, it gives the player a salary. It also demonstrates a use of a writeable property as this line would not work without the set section in the Player’s Cash property.</p> <p>This syntax might be considered more readable than an equivalent such as:</p>

#	Description
	<code>currentPlayer.ReceiveCash(SALARY);</code>
16	<p>Step 6 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>We are also replacing the behavior for PassOver, the approach is the same. We could write both of these in a common method and call that common method.</p>

**Update: Board.cs**

#	Description
3	<p><code>myStart = new StartingLocation("Start");</code></p> <p>Step 7 of “Background: Steps for Polymorphism” starting on page 44.</p> <p>The only change to board is to create a new kind of BoardLocation for the starting location. Notice that while we are creating an instance of the StartingLocation class, we are assigning it to a reference of the type BoardLocation.</p> <p>Any derived classes of BoardLocation can stand in for a BoardLocation object. That is, a base class can be substituted with a derived class. A reference to a base class may refer to an instance of the base type or an instance of a derived type.</p> <p>The static or compile time type of the myStart attribute is BoardLocation. The dynamic or run-time type of the myStart attribute happens to be StartingLocation. The static type of a variable, reference, property never changes. The dynamic type can vary while the program runs.</p> <p>Consider the Player class. It holds a reference to its current location. The static type of that reference is always BoardLocation. The Player can only send messages to that object based on the definition of BoardLocation. At times during the execution of the program, the dynamic type of the player’s location attribute will change as the player moves around the board.</p>

**Update: Player.cs**

#	Description
2	<p><code>private int myCash;</code></p> <p>Add a new attribute to the Player. The requirements have changed and we need to change along with them. In reality we are tackling this problem iteratively so it is only natural that classes will change over time. Eventually we hope to see some stability in our classes.</p>
5	<p><code>public Player(string name, BoardLocation startLoc, int startCash) {</code></p> <p>The constructor now takes a third parameter, startCash. This sets the Player’s initial balance. This is a characteristic of the player but the initial balance is a rule of the game. By making it a parameter to the constructor, we are able to easily handle this apparent split of responsibility.</p>
6 – 8	<p>Perform parameter validation. Our policy regarding parameter checking is to validate in constructors and all the public methods in Game. We validate that name and startingLocation are not null. We also validate that the starting cash is positive.</p>
24	<p><code>set { myLocation = value; }</code></p> <p>In the first iteration the player simply moves around the board, so the Player’s Location property was read only. Making the property read/write allows BoardLocation objects like the SpringBoard to move the player.</p>
16	<p><code>public int Cash {</code></p> <p>Add a Property to support the cash attribute. StartingLocation uses this property to give cash to the Player in response to the Player landing on or passing over it.</p>
28 – 40	<p><code>public void TakeATurn(PairOfDice dice) {</code></p> <p>We almost entirely rewrote this method. There are two major changes:</p> <ul style="list-style-type: none"> <li>▪ The output now takes more than one line, allowing a BoardLocation to add to the output logically.</li> <li>▪ The player now sends the LandOn method and the PassOver method.</li> </ul> <p>After compiling and executing this code, you will see output like the following:</p> <p>Player#1 on BL(Loc#12) has 1500</p>

#	Description
	<pre>rolls 3 will land on BL(Loc#15) and now has 1500</pre> <p>If a BoardLocation prints out a line because a Player landed on it, the output will appear between the third and fourth lines. If a BoardLocation prints out a line because a Player passed over it, the output will appear before the third line.</p> <pre>Player#0 on BL(Loc#32) has 1500 rolls 9 Player#0 passed over Start and receives 200 will land on BL(Loc#1) and now has 1700</pre> <p>We will eventually take all of the output of our classes and move to using Events.</p>
32 – 38	<pre>myLocation = Location.Next;</pre> <p>If a Player happens to be on the StartingLocation, then the Player will receive 200 for leaving this location unless we skip past it before sending the PassOver message. If you count the total number of BoardLocation objects a Player passes, the count is correct. We leave the current location and then send PassOver to the next Location. We continue doing this until the last BoardLocation. The last BoardLocation does not receive the PassOver message but instead receives the LandOn method.</p>

**Update: Game.cs**

#	Description
9	The constructor for the Player class now requires three parameters instead of one.

**Object Collaboration**

There are two significant changes to classes from iteration 1. One is the Player.TakeATurn method. The second is the Board's constructor, which creates an instance of StartingLocation for myStart instead of a BoardLocation object. We do not show the collaboration for the Board's constructor.

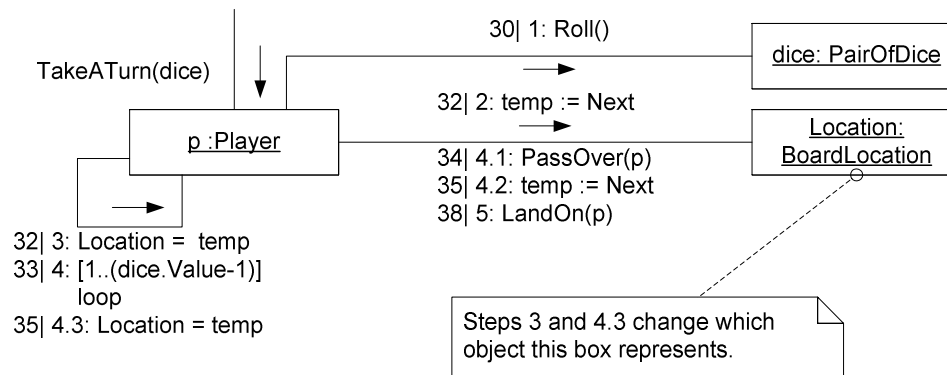
The turn begins as it did in iteration 1. The change is during the loop. The Player first leaves its current location by getting the Next property of its myLocation attribute. The Player then reassigns its myLocation attribute to the value returned from the Next property.

The player continues by moving 1 to the value of the dice (minus 1), sending the message PassOver to the object referred to the Player's myLocation attribute, and then moving to the next location.

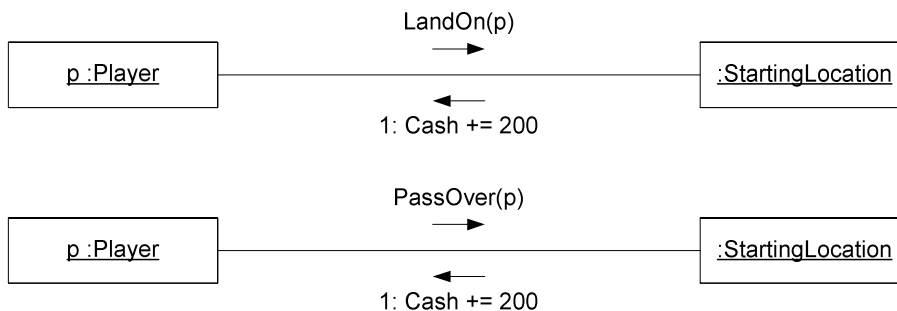
Finally, when the player is on its destination, it sends the message LandOn to the object referred to by the Player's myLocation attribute.

The Player leaves its current location before sending the first PassOver to avoid sending LandOn in the previous turn and PassOver in the next turn. Consider a Player landing on StartingLocation in the previous turn. They receive 200 for this event. If, in the next turn, they send PassOver to their current location (StartingLocation), they would receive an additional 200.

The following diagram shows the collaboration diagram equivalent of the code in the updated version of Player.cs. In addition to following the convention of numbering messages from 1, we include the line number from "Updates to: Player.cs" on page 38 .

**Figure 19: UML – Collaboration: Player.TakeATurn(PairOfDice)**

When a Player happens to land on or pass over an instance of StartingLocation, the Player receives 200. We can show this as a special case of the LandOn and PassOver methods for StartingLocation:

**Figure 20: UML – Collaboration: StartingLocation.LandOn(Player)/PassOver(Player)**

## Background: Steps for Polymorphism

We want different board locations to respond to the same requests with different responses. C# requires 8 steps to make this happen:

1. Create or modify a class to serve as a base class.

```
public class BoardLocation {
```

2. Add methods to the base class that the subclass is meant to override.

3. Make sure those methods targeted for overriding are virtual.

```
public virtual void LandOn(Player currentPlayer) {}
```

4. Create a subclass of that base class.

5. Make sure that the subclass explicitly lists the base class as its parent.

```
public class StartingLocation : BoardLocation {
```

6. Override the methods from the base class in the derived class.

```
public override void LandOn(Player currentPlayer) {
    currentPlayer.Cash += SALARY;
}
```

7. Create instances of the derived class.

```
myStart = new StartingLocation("Start");
```

8. Make sure that the new virtual methods are actually called from somewhere.

```
for(int i = 1; i < dice.Value; ++i) {
    Location.PassOver(this);
    Location = Location.Next;
}
Location.LandOn(this);
```

## Build 2: The Remaining Classes

While we had to follow eight steps the first time we introduced polymorphism, now we only have to follow three. Those three steps are:

1. Create a new subclass of the base class.
2. Override the methods in the base class that should have different behavior. We will add:
  - SpringBoard
  - SurpriseBill
3. Make sure instances of the new class get created and inserted into the system properly.
  - We will update the constructor in the Board class since that is the class that creates all BoardLocation objects.

### File: SurpriseBill.cs

```
1| using System;
2| namespace BoardGame {
3|     public class SurpriseBill : BoardLocation {
4|         private int myFine;
5|
6|         public SurpriseBill(string name, int fine) : base(name) {
7|             Assert.Argument(fine > 0, "fine", fine, "should be > 0");
8|
9|             myFine = fine;
10|        } // end SurpriseBill constructor
11|
12|        public override void LandOn(Player p) {
13|            p.Cash -= myFine;
14|            Console.WriteLine("\t{0} pays {1}", p.Name, myFine);
15|        } // end LandOn method
16|    } // end SurpriseBill class
17| } // end BoardGame namespace
```

### File: SpringBoard.cs

```
1| using System;
2| namespace BoardGame {
3|     public class SpringBoard : BoardLocation {
4|         private const int MAX_PASS_COUNT = 13;
5|         private const int COUNT_MULTIPLIER = 3;
6|         private int passCount = 0;
7|
8|         public SpringBoard(string name) : base(name) {
9|             } // end SpringBoard constructor
10|
11|         public override void LandOn(Player p) {
12|             BoardLocation dest = CalculateDestination();
13|             p.Location = dest;
14|             passCount = 0;
15|             Console.WriteLine("\tPlayer sprung to: {0}", dest.Name);
16|        } // end LandOn method
17|
18|        public override void PassOver(Player p) {
19|            if(passCount < MAX_PASS_COUNT) {
20|                ++passCount;
21|            } // end if
22|            Console.WriteLine("\tSpringBoard pass count: {0}", passCount);
23|        } // end PassOver method
24|
25|        private BoardLocation CalculateDestination() {
26|            BoardLocation dest = this;
27|            for(int i = 0; i < COUNT_MULTIPLIER * passCount; ++i) {
28|                dest = dest.Next;
29|            } // end for loop
```

```
30|         return dest;
31|     } // end CalculateDestination method
32| } // end SpringBoard class
33| } // end BoardGame
```

### File: Board.cs

```
1| namespace BoardGame {
2|     public class Board {
3|         private BoardLocation myStart;
4|
5|         public Board() {
6|             myStart = new StartingLocation("Start");
7|
8|             BoardLocation current = StartingLocation;
9|             BoardLocation newNext = null;
10|
11|             for(int i = 1; i < 40; ++i) {
12|                 switch(i) {
13|                     case 2:
14|                         newNext = new SurpriseBill("Leak in Roof", 100);
15|                         break;
16|                     case 19:
17|                         newNext = new SpringBoard("SpringBoard");
18|                         break;
19|                     case 26:
20|                         newNext = new SurpriseBill("Minor Cooking Incident", 75);
21|                         break;
22|                     default:
23|                         newNext = new BoardLocation("Loc#" + i);
24|                         break;
25|                 } // end switch
26|                 current.Next = newNext;
27|                 current = newNext;
28|             } // end for loop
29|             current.Next = StartingLocation;
30|         } // end Board constructor
31|
32|         public BoardLocation StartingLocation {
33|             get { return myStart; }
34|         } // end StartingLocation property
35|     } // end Board class
36| } // end BoardGame namespace
```

## Build & Execute Instructions

Follow these instructions to see the change to your system:

1. Create the SurpriseBill.cs file.
2. Create the SpringBoard.cs. file.
3. Update the Board.cs file. Change the constructor to use a switch statement.
4. Compile and execute your program. Redirect the output to a file:

```
c:\C#\BoardGame>MainApp> gameresults.txt
```

If you run this enough you will be able to see each of the different kinds of BoardLocation objects take part in changing what happens to each Player the Player moves around the board.

## Code Explained

### File: SurpriseBill.cs

#	Description
1	We perform output. Avoid having to fully qualify the Console class.

#	Description
3	<code>public class SurpriseBill : BoardLocation {</code> Create a new class, called SurpriseBill, which inherits from BoardLocation.
6	<code>public SurpriseBill(string name, int fine) : base(name) {</code> This constructor takes two parameters. The first one is passed to the constructor in the base class, which simply stores the name of the BoardLocation. The constructor in SurpriseBill records the second parameter in an attribute.
12	<code>public override void LandOn(Player p) {</code> Replace the default behavior inherited from BoardLocation on what to do when landed upon by a player. Note that the method signature is the same. The only change to the signature is that we must use the override keyword.
13	<code>p.Cash -= myFine;</code> Update the Player's balance. Notice that while this looks like we are directly accessing an attribute, this actually uses the setter of the Cash property.
14	<code>Console.WriteLine("\t{0} pays {1}", p.Name, myFine);</code> Produce output so we can see that this actually happens.

**File: SpringBoard.cs**

#	Description
3	<code>public class SpringBoard : BoardLocation {</code> SpringBoard is a BoardLocation.
6	<code>private int passCount = 0;</code> An attribute to keep track of the running total of Players that have passed the spring board.
8	<code>public SpringBoard(string name) : base(name) {</code> Define a constructor that simply takes a name parameter and passes it up to the base class. Notice that we do not validate any parameters. The base class validates the name parameter, which is our only parameter.
11	<code>public override void LandOn(Player p) {</code> Replace the default behavior of LandOn with custom behavior.
12	<code>BoardLocation dest = CalculateDestination();</code> Use a private method to calculate the final destination, which is passCount * 3 spaces ahead.
13	<code>p.Location = dest;</code> Having calculated the new destination, change the Player's location. Notice that while this looks as if we are directly accessing an attribute, this code actually calls the setter of the Location property. The setter verifies that the provided BoardLocation is not null.
14	<code>passCount = 0;</code> The requirements say that after springing a player ahead, reset the count back to 0.
15	<code>Console.WriteLine("\tPlayer sprung to: {0}", dest.Name);</code> Display a message so we can verify that this actually happens in the output.
18	<code>public override void PassOver(Player p) {</code> Change the default behavior for PassOver. Increment our counter, up to a maximum of 13. Display a message so we can verify that the pass count is changing.
25	<code>private BoardLocation CalculateDestination() {</code> This is a support method. It simply starts from the current square, accessible via the "this" variable, and uses the Next property until determining the final location.

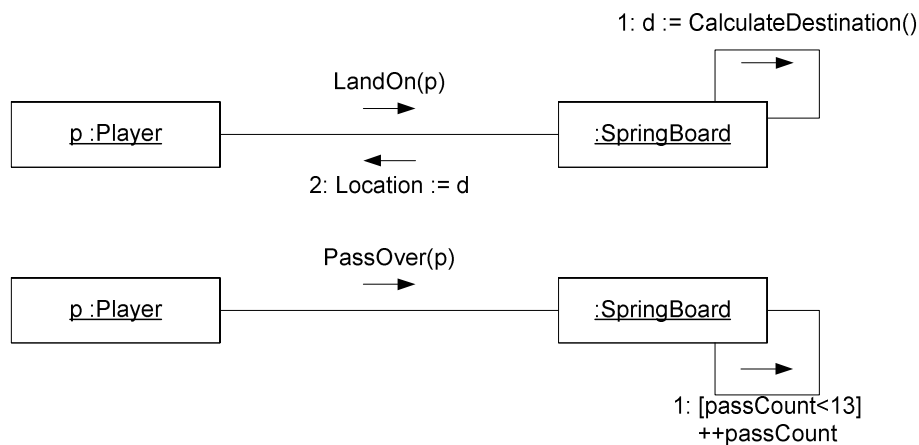
**File: Board.cs**

#	Description
12	<code>switch(i) {</code> Based on the index, create a different kind of BoardLocation. This is a simple way to create different kinds of BoardLocation objects. Using a switch might not seem "object-oriented" and it might not be. However, we do not yet have objects and we have to create them.

#	Description
	Using switch statements with existing objects is dubious. Using a switch statement either before an object exists or when an object is about to transition between the system and outside of the system always requires special coding in systems that do not support polymorphism on class methods (like Smalltalk). We will replace this code by reading from a file. When we do so, we will still use a switch. There are good design patterns to solve this problem. However, we are not going to use this in this project because they are overkill for the problem we are trying to solve.
13, 16, 19,	<code>newNext = new XXX(...);</code> The same variable, newNext, is set to one of three different kinds of classes depending on the case. The same variable can refer to different types of objects. In reality the flexibility is limited to any class which is derived from (or a subclass of) BoardLocation. The static type of the newNext variable is BoardLocation. During the execution of this loop. The dynamic type is BoardLocation when i is anything but 2, 19, or 26. When i is 2 or 26, the dynamic type is SurpriseBill. When i is 19, the dynamic type is SpringBoard. The remainder of the loop stays the same.

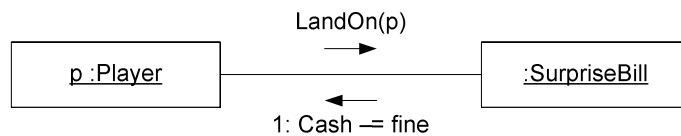
## Object Collaboration

SpringBoard does something for both passing over it and landing on it:



**Figure 21: UML – Collaboration: SpringBoard.LandOn(Player)/PassOver(Player)**

SurpriseBill only does something when a Player lands on it:



**Figure 22: UML – Collaboration: SurpriseBill.LandOn(Player)**

## Review

- Inheritance allows the definition of one class, the subclass or derived class, be defined in terms of changes to another class, the base class or super class.
  - Inheritance is a very strong relationship between classes. Use it sparingly.
  - Changes in the base class might break subclasses.



- The base class and derived classes might change in incompatible ways over time.
- Read about the Liskov substitution principle in [6].
- Constructors are not inherited in the same way that other methods are inherited. The derived class constructor can use the base class constructor but the base class constructor is not a part of the interface (available methods) of the derived class.
- When we construct an instance of a derived class, each of the base class constructors are called, top to bottom, before finally executing the object's constructor.
- Typically a subclass will define constructors that take in information stored in the base class and information unique to the derived class. The information that belongs in the base class is sent up to the base class constructor by adding “: base(parameters)” after the method signature of the constructor.
- If a base class does not have a zero-argument constructor, a derived class must add a constructor that calls a base class constructor.
- A reference has a static type and a dynamic type.
  - The static type is determined by looking at how it is declared. We can tell at compile time what a reference's static type is.
  - The dynamic type depends on the particular object to which a reference points.
  - A reference cannot point to just any object, it can only point to instances of its static type or instances of classes that are subclasses of its static type.
- Polymorphism allows a message sent to an object to invoke different methods, based on the class of the receiving object.
  - Polymorphism as we are using it requires inheritance.
  - Polymorphism in C# requires the use of the virtual keyword in the base class.
  - Polymorphism in C# requires the use of the override keyword in the derived class.
  - The selection of messages which can be sent is limited to those defined in the static type of the reference.
  - The selection of which method gets executed is determined by the dynamic type of the reference.
  - A derived class can override any number of the virtual methods in the base class. Zero is acceptable, although it is dubious.
  - Polymorphism is only available on non-static methods. This is not intrinsic to object-oriented programming; rather it is the way C# is designed (as well as most commercial object-oriented languages).
- There are several things required to get polymorphism:
  - Create or modify an existing class to be a base class. It should have some virtual features (there are other options; we have not seen these yet). BoardLocation is our example.
  - Create subclasses and override some of the virtual features in the base class. StartingLocation, SurpriseBill and SpringBoard are examples.
  - Create instances of the derived classes. We do this in the Board constructor.
  - Some object must actually use the virtual methods. The Player sends both LandOn and PassOver to BoardLocation objects.
  - Before objects exist it is hard to get polymorphism in most object-oriented languages. Smalltalk is a notable exception.

- On the transition from in a system to out of a system, Polymorphism often does not work. It can, but doing so has many implications.
- Properties can be read only or read write. The code for the get section or set section can be simple or complex.
- We can validate input parameters and throw exceptions if something is wrong with them. Doing so makes it easier to find problems sooner to the source and remove them. We can do more, see [6] for more about programming by contract.

### **Final Observation**

We added two new kinds of classes and updated the Board class. We now have entirely different behavior and most of the code stayed the same. We have a kind of BoardLocation that moves the Player forward a number of locations based on how many times other players have passed it. This is quite different from the first iteration where players moved around the board. It is also quite different from the other locations that add or subtract money.

## Iteration 2a: Reading from a File

This section adds no new functionality but it will make constructing the board a little less tedious. Rather than hardcode the board size and the board locations, we will instead store the information in a file and read the file.

### The Problem

We have the information for a number of BoardLocation objects stored in a file. The file is a simple text file, where each BoardLocation is listed on its own line. A given line has several pieces of information, separated by a comma “,” character. The order of the locations in the file is the same as the order of the board.

Warning, this file contains information we will not use until future iterations of the project. Rather than give you a partial file or make you update the file, we instead present a complete file and make sure to write the code so it can handle the parts of the file for which we do not have a complete solution.

### File: BoardLocations.txt

Note we show two columns to take up less space, this is meant to be read top to bottom on the left then top to bottom on the right.

Start, Starting Location	spring, Spring Board
city, Iowa City, 60, 2, Iowa	city, Rockford, 220, 18, Illinois
random, Random Effect	random, Random Effect
city, Des Moines, 60, 4, Iowa	city, Chicago, 220, 18, Illinois
surprise, Leak In Roof, 100	city, Springfield, 240, 20, Illinois
transportation, Bus, 200	transportation, Taxi, 200
city, Dallas, 100, 6, Texas	city, Yakima, 260, 22, Washington
random, Random Effect	city, Redmond, 260, 22, Washington
city, Houston, 100, 6, Texas	utility, Green Electricity, 150
city, Austin, 120, 8, Texas	city, Olympia, 280, 22, Washington
bnb, Knights Arms, 25	bnb, Sheltered Arms, 25
city, Scottsdale, 140, 10, Arizona	city, Newport, 300, 26, Vermont
Utility, Natural Gas, 150	city, Bennington, 300, 26, Vermont
city, Sedona, 140, 10, Arizona	random, Random Effect
city, Phoenix, 160, 12, Arizona	city, Montpelier, 320, 28, Vermont
transportation, Rail, 200	transportation, Air, 200
city, Alexandria, 180, 14, Louisiana	random, Random Effect
random, Random Effect	city, San Francisco, 400, 50, California
city, New Orleans, 200, 16, Louisiana	surprise, Minor Cooking Incident, 75
city, Baton Rouge, 200, 16, Louisiana	city, Sacramento, 400, 50, California

The actual text file has 40 lines. The first thing on every line is a unique string identifying the kind of class the line represents. For example, “start” is for StartingLocation and “spring” is for SpringBoard. There are several other types for which we have no implementation. Until we have an implementation for the class, we will continue to use BoardLocation.

The next item on each line is the name to use for the BoardLocation. After the name, each kind of line contains its own unique characteristics. For example, the “surprise” type has 1 additional field, amount, giving it a total of 3 fields. During the third iteration we will be taking care of the city, utility and transportation types.

### A Solution

To build the board from this file, we need to do three things:

1. Read the file one line at a time.
2. Parse the individual parts (tokens) within the line.
3. Construct the right kinds of BoardLocation based on information contained within the line.

## Read lines from a file

Reading a file in C# can be quite easy, we use one of the classes provided in the .NET framework to open a file and read it one line at a time. The class that we will use is in the namespace System.IO. Its name is StreamReader, and here is a simple example of its use:

```
1| using System.IO;
2| StreamReader reader = new StreamReader("BoardLocations.txt");
3| string line = null;
4| while((line = reader.ReadLine()) != null) {
5|     Console.WriteLine(line);
6| } // end while loop
7| reader.Close();
```

### Example

#	Description
1	StreamReader is in the System.IO namespace. Use that namespace so we do not have to use the fully-qualified class name.
2	Declare a StreamReader and initialize it by creating a StreamReader on the provided filename. If the file name is not found, this will throw a System.IO.FileNotFoundException.
3	Declare a temporary variable to hold each line as we read it in.
4	Get the next line and store it in the line variable. If line is assigned the value of null, we have reached the end of the file.
5	Print the line we just read in. Give the example file, the output will be (only first 3 lines shown): <pre>start, Starting Location city, Iowa City, 60, 2, Iowa random, Random Effect</pre>
7	Remember to Close() all files. If you do not, they will not be properly cleaned up. Even though C# will remove the object from memory through garbage collection, the StreamReader class uses resources from the operating system, namely file handles. Our application will keep those around unless you explicitly close files.  Later on we will see an idiom using a try {} finally {} combination to guarantee that resources get properly cleaned up. Coming up we will take care of this problem without using try {} finally {} because we will instead write a class to handle our file IO.

## Parse strings into tokens

Next, we need to convert one of the lines into separate tokens. There are several ways to do this in .NET, luckily it is quite easy for a simple example like this. The string class has the Split method to break a line into parts. It takes in an array of characters, which it uses to break the line into parts. It returns an array of strings; the original line split into parts. Here is a simple example:

```
1| using System;
2| const char[] SPLIT_CHARS = new char[] { ',' }
3| string line = "city,Iowa City,60,2,Iowa"
4| string[] tokens = line.Split(SPLIT_CHARS);
5| for(int i = 0; i < tokens.Length; ++i) {
6|     Console.WriteLine("tokens[{0}] = {1}", i, tokens[i]);
7| }
```

### Example

#	Description
2	Define an array of characters with only one entry, a comma. This is the parameter we will pass into the Split() method.
3	Define a line to parse.
4	Declare an array of strings called tokens. Initialize it to the result of calling the Split method on

#	Description
	line. The result should be an array whose length is 5.
5	Print the array. The output for this example will be:
–	tokens[0] = city
6	tokens[1] = Iowa City
	tokens[2] = 60
	tokens[3] = 2
	tokens[4] = Iowa

## Constructing the right kinds of objects

Finally, we need to construct the right kind of class based on the type field, which is the first field on each line. We do not handle all kinds of types yet, so we will create a complete switch statement, but some of the lines in the switch statement simply create a BoardLocation.

Here is an example that creates the right kind of BoardLocation based on the type:

```

1| string[] tokens = line.Split(SPLIT_CHARS);
2| string type = tokens[0];
3| string name = tokens[1];
4| BoardLocation current = null;
5|
6| switch(type) {
7|     case "start": current = new StartingLocation(name); break;
8|     case "city":  current = new BoardLocation(name); break;
9|     case "random": current = new BoardLocation(name); break;
10|    case "surprise":
11|        int fine = Int32.Parse(tokens[2]);
12|        current = new SurpriseBill(name, fine);
13|        break;
14|    case "transportation": current = new BoardLocation(name); break;
15|    case "bnb":            current = new BoardLocation(name); break;
16|    case "utility":        current = new BoardLocation(name); break;
17|    case "spring":         current = new SpringBoard(name); break;
18|    default:
19|        throw new
20|            ApplicationException("Unknown BoardLocation Type: " + type);
21| } // end switch

```

## Example

#	Description
1	This example builds on the previous two. Assume you've opened a file and read a line. Now you split the line into individual tokens. You store the type, tokens[0] in one temporary variable and the name of the location, tokens[1], in another. Every line in the file is guaranteed to have at least these two fields (assuming the file is well-formed).
–	
3	
4	Define a temporary variable to hold the soon-to-be-created BoardLocation. Remember that a reference to a base type may refer to instances of the base type or any derived class. The current variable may refer to BoardLocation, StartingLocation, SurpriseBill or SpringBoard.
6	Select one of the following sub-sections based on the type temporary variable. We have already seen a switch statement during the second iteration. This one uses strings instead of integers. Also, C# requires that every case exits the switch statement, either using break, return or throw. The switch statement in C# is really a nicer way of writing a series of if-then-else statements.
7	If the type is "start", construct an instance of the Start. Note we must use something to force control out of the switch statement. You have three options: break, return, throw.
8	If the type is "city", construct an instance of BoardLocation. We do this because we have not yet created a class to handle this type. As we support new kinds of BoardLocation objects, we will update individual cases in this switch statement to construct the right kinds of board location objects.
–	
17	
19	The default: clause is optional. In this case we want to fail if we find a type we do not know

#	Description
	about. In a strict sense, the .NET recommendation is to create your own Exception class, which inherits from ApplicationException. That is a bit much for now, so we just use ApplicationException and provide a reasonable string to assist in debugging the problem.

## Putting It All Together

We can just put all of the above code directly into the constructor of the Board class, or maybe use some private methods to make everything a bit easier to follow. We are instead going to create a class called BoardLocationReader that puts all of this together.

We are doing this for two reasons. First, it keeps board simple and less coupled to the how we decided to store the BoardLocation objects. Second, we will use the BoardLocationReader as a utility class later on when we are experimenting with ADO.NET.

This class is rather more complex than the above examples might imply for three reasons:

- First it is a complete class.
- Second, because we are using file IO, we need to be more careful about cleaning up resources we've allocated.
- Third, we are writing the interface of this class to look like a System.Collections.IEnumerator. It is not important what an IEnumerator is or does for now. It will come up later on and we will have this concrete example from which to work.

### File: BoardLocationReader.cs

```
1| using System;
2| using System.IO; // we use StreamReader, which is in this namespace
3| namespace BoardGame {
4|     public class BoardLocationReader {
5|         private static char[] SPLIT_CHARS = new char[] { ',' };
6|         private string suppliedFileName;
7|         private StreamReader reader;
8|         private BoardLocation current;
9|
10|         public BoardLocationReader(string fileName) {
11|             Assert.NotNull(fileName, "fileName");
12|
13|             suppliedFileName = fileName;
14|             Reset();
15|         } // end BoardLocationReader constructor
16|
17|         ~BoardLocationReader() {
18|             Close(); // clean up resources
19|         } // end BoardLocation destructor
20|
21|         public void Close() {
22|             if(reader != null) {
23|                 reader.Close();
24|                 reader = null;
25|             } // end if
26|         } // end Close method
27|
28|         public void Reset() {
29|             Close();
30|             reader = new StreamReader(suppliedFileName);
31|         } // end Reset method
32|
33|         public bool MoveNext() {
34|             string line = GetNextLine();
35|             if(line == null) {
36|                 current = null;
37|                 return false;
```

```

38|         } // end if
39|         SetCurrentFrom(line);
40|         return true;
41|     } // end MoveNext method
42|
43|     public BoardLocation Current {
44|         get { return current; }
45|     } // end Current property
46|
47|     private string GetNextLine() {
48|         string line = null;
49|         do {
50|             line = reader.ReadLine();
51|             if(line != null) {
52|                 line = line.Trim();
53|             }
54|         } while(line != null && line.Equals(""));
55|
56|         return line;
57|     } // end GetNextLine method
58|
59|     private void SetCurrentFrom(string line) {
60|         string[] tokens = line.Split(SPLIT_CHARS);
61|         string type = tokens[0];
62|         string name = tokens[1];
63|
64|         switch(type) {
65|             case "start": current = new StartingLocation(name); break;
66|             case "city":  current = new BoardLocation(name); break;
67|             case "random": current = new BoardLocation(name); break;
68|             case "surprise":
69|                 int fine = Int32.Parse(tokens[2]);
70|                 current = new SurpriseBill(name, fine);
71|                 break;
72|             case "transportation": current = new BoardLocation(name); break;
73|             case "bnb":            current = new BoardLocation(name); break;
74|             case "utility":        current = new BoardLocation(name); break;
75|             case "spring":         current = new SpringBoard(name); break;
76|             default:
77|                 throw new
78|                     ApplicationException("Unknown BoardLocation Type: " + type);
79|         } // end switch
80|     } // end SetCurrentFrom method
81| } // end BoardLocationReader class
82| } // end BoardGame namespace

```

We want to test this works before updating the Board class. We can simply add a few lines to our MainApp.cs class. If we try to add a Main to this class, here's what will happen:

1. You will try to compile using: `csc *.cs`
2. The compiler will complain that there is more than one Main.

You have three options. First, you can change MainApp.cs. Second you can create a dll file and use it during compilation. Third, you can put a Main in BoardLocationReader or in another class that is equivalent to MainApp but has a different name.

We have not yet used assemblies directly. We will but not until much later so we will rule that out. If you do want to put a Main() in If you do, you will have to use the following command to compile:

```
csc BoardLocationReader.cs BoardLocation.cs Die.cs PairOfDice.cs Player.cs ...
```

You do not need Board.cs and you cannot include MainApp.cs. We assume you'll choose to update MainApp.cs.

### Changes to: MainApp.cs

```

1| // add the following lines at the beginning of the Main method
2| BoardLocationReader r = new BoardLocationReader("BoardLocations.txt");

```

```

3| while(r.MoveNext() == true) {
4|     Console.WriteLine(r.Current);
5| } // end while loop

```

#	Description
2	Create an instance of the BoardLocationReader using the file BoardLocations.txt. If the file is not in the current directory, this line will generate a FileNotFoundException.
3	When you create the BoardLocationReader, it is just before reading the first BoardLocation. Using MoveNext causes the first location to be read and stored in the Current property. If there are no more BoardLocation objects to be read, MoveNext returns false, otherwise it returns true.
4	Print the Current Property, which is an instance of a BoardLocation or one of its subclasses.

### File: BoardLocationReader.cs

Now that we have an idea of how to use BoardLocationReader (lines 7 – 10 of MainApp.cs), we will describe all of its gory details.

#	Description
2	We are using StreamReader, which is in the System.IO namespace.
5	Define the characters upon which we will split input lines. An array of chars, size 1, with ‘,’ in the 0 <sup>th</sup> position.
6	We will store the name of the file name provided to the constructor. This allows us to reopen the file and read it again. We want to have this as long as the BoardLocationReader is around so it must be an attribute.
7	This is the StreamReader we use throughout the reading of the file. We use it across methods and it needs to be kept between invocations of the MoveNext() method, so it must be an attribute.
8	The current attribute holds storage for the Current property. This attribute holds the most recently read BoardLocation. Calls to MoveNext() reset current to either the newly read in BoardLocation or null if there are no more BoardLocation objects to be read. It starts with null because the first BoardLocation should not be available until after the first call to MoveNext(). This is based on the design of the IEnumerable interface. We will be seeing more of this later.
13	Store the parameter in an attribute. If we forget to do this before calling Reset(), our application will throw an exception.
14	Put the object in a well defined state. The Reset() method can be used by a client to start reading from the beginning. This works well for construction as well as after construction, so simply use the Reset() method to prepare for the first call of MoveNext(). By calling Reset() we have opened a file. Now we are responsible for closing that file.
17	<code>~BoardLocationReader() {</code> Define a destructor. Destructors are called automatically when an object goes out of scope and is garbage collected. We have this to ensure that if someone creates an instance of BoardLocationReader but does not use the Close() method, that we will eventually close the file anyway. The user does not have to use Close(). If they do, unneeded resources will be cleaned up sooner rather than later.
18	Clean up any resources we have allocated. In this case we simply close the StreamReader. Since we need to do this in different places, we write it as a method to avoid duplication.
21	Allow someone to explicitly close the file used by the BoardLocationReader. This potentially opens us up to a NullPointerExpection. If someone calls Close() then MoveNext() we will get a NullPointerExpection. We could guard for this but the code is already complex enough.
22	The reader may not be initialized. Check before sending the Close() message to the reader. This situation arises when Reset() is called from the constructor. The constructor has only set the name of the file to be opened. It then calls Reset(). The reader attribute is null. So we need to



#	Description
	make sure that it is appropriate to try and close the file.
23	Actually close the file.
24	Reset the reader attribute back to null. This actually releases the memory used by the reader so that it may be garbage collected.
28	Reset() puts the BoardLocationReader back to the beginning of the file. Calling Reset() followed by MoveNext() gives the first BoardLocation in the file.
29	First Close() the file.
30	Now reopen it.
33	MoveNext() will move to the next available BoardLocation. If there was another one, it returns true. If not, it returns false.
34	Read the next line from the StreamReader.
35	GetNextLine() returns null if there was not a file to be read, this mimics the behavior of the ReadLine method on StreamReader.
36	Since there was not another BoardLocation, set current to null so we do not send a mixed message.
37	Return false so the user of MoveNext knows that there are no longer any BoardLocation objects available
39	There was another BoardLocation, use the current line to construct the next BoardLocation.
40	Return true to let the caller know that the Current property now points to the newly acquired BoardLocation object.
43	Define a public, read-only property called Current. This is the thing just read during MoveNext().
47	Acquire a line. This performs some additional checking to ignore blank lines. It is a bit more robust than just calling ReadLine on the reader directly. We made this a method to keep individual methods short.
48	Define a temporary variable to store the line about to be read.
49	We might get a blank line and need to read the next line. We need a loop. We know we are going to try and read at least one line, so we use a do loop. Do loops are guaranteed to execute at least once.
50	Read the next line.
51 – 52	If the line is not null, use the Trim() method and store the result. Trim() returns a new string with the leading and trailing white space (tabs and spaces) removed. The original string is unchanged. <i>Note: string objects are immutable, so the only option is for Trim() to create a new string and return it.</i>
54	If the line is not null and it is a blank line, read another line. Keep reading until we have hit the end of the file or found a non-blank line.
56	Return the line we've successfully read. Note that this value could be null if we hit the end of the file. So this mimics the behavior of ReadLine with the added benefit of ignoring blank lines. The blank lines are mostly an issue at the end of the file.
59	We've successfully read a line, convert that line into an instance of BoardLocation or one of its subclasses.
60 – 75	Split the line and create a type and name temporary variable. This code should look familiar since it is nearly the same as the switch statement we used earlier. The only difference is that we have a current attribute instead of a current temporary variable.
68	The "surprise" type refers to SurpriseBill. SurpriseBill has three tokens: type, name and fine amount. We need to convert the third parameter to an integer before passing it into the

#	Description
	<p>constructor of SurpriseBill.</p> <p>We have a string available from the array of tokens. We could just pass that string into SurpriseBill and make it convert the string to an integer. We do not do this because the fact that we have a string has to do with how we decided to store the BoardLocation objects in a file. If we instead stored them in a database, we would instead get back an integer. The BoardLocationReader is the expert on how to deal with the peculiarities of the storage mechanism.</p>
76	Notice that our default section does not have a break in it. Every section within a switch must
–	explicitly exit. The normal way is to use break. There are two more options, use return or
78	throw. If we actually put a break after the throw, the C# would warn of unreachable code.

## Build & Execute Instructions

To build these three files, do the following:

1. Create the BoardLocationReader.cs file.
2. Update the MainApp.cs file.
3. Compile the application:

```
c:\C#\BoardGame>csc *.cs
```

4. Execute the Program, providing parameters to make the game quick.

```
c:\C#\BoardGame>MainApp 1 1
StartingLocation(Starting Location)
BoardLocation(Iowa City)
BoardLocation(Random Effect)
BoardLocation(Des Moines)
SurpriseBill(Leak In Roof)
BoardLocation(Bus)
BoardLocation(Dallas)
BoardLocation(Random Effect)
BoardLocation(Houston)
BoardLocation(Austin)
BoardLocation(Knights Arms)
BoardLocation(Phoenix)
BoardLocation(Natural Gas)
BoardLocation(Sedona)
BoardLocation(Scottsdale)
... ETC. ...
```

## Using BoardLocationReader in Board

Now that we have a working class to read BoardLocation objects, we need to update Board.cs to use it. We will not make the board responsible for knowing about how to create a BoardLocationReader. Instead, the Game will create it and pass it in to the Board's constructor.

Using BoardLocationReader will require two changes, one to the Board constructor and one to the Game. The following only shows those changes, rather than the whole class.

### Update to: Board.cs

```
1| public Board(BoardLocationReader reader) {
2|     Assert.NotNull(reader, "reader");
3|
4|     reader.MoveNext();
5|     myStart = reader.Current;
6|
7|     BoardLocation current = StartingLocation;
8|     BoardLocation newNext = null;
```

```

9|
10|     while(reader.MoveNext() == true) {
11|         newNext = reader.Current;
12|         current.Next = newNext;
13|         current = newNext;
14|     }
15|     current.Next = StartingLocation;
16| }

```

### Update to : Game.cs

```

1| // update the Board attribute, remove initialization
2| private Board myBoard;
3|
4| // update the Game constructor
5| public Game(int playerCount) {
6|     // parameter checking not shown
7|     BoardLocationReader r = new BoardLocationReader("BoardLocations.txt");
8|     myBoard = new Board(r);
9|     r.Close();
10|    // remainder of the constructor is the same
11| }
12| }

```

## Code Explained

### File: Board.cs

#	Description
1	Change the signature of the Board constructor to take in a BoardLocationReader object. Ultimately we need to create a BoardLocationReader and provide it a file name. If we later create a new kind of BoardLocationReader, e.g. one that brings in BoardLocation objects from a data base, the Game can create that one instead and the Board is unchanged. So the Board is open to changes in how the system acquires BoardLocation objects. It is closed from how it forms the board connections in memory.
4	The reader starts just before the first object. Move it to the first BoardLocation object.
5	Get the first BoardLocation object from the BoardLocationReader and store it as the starting location. We get the most recently read object by using the Current property.
7 – 15	<p>The remainder of the constructor looks much like the original version of Board.cs:</p> <pre> BoardLocation current = StartingLocation; BoardLocation newNext = null;  for(int i = 1; i &lt; 40; ++i) {     newNext = new BoardLocation("Loc#" + i);     current.Next = newNext;     current = newNext; }  current.Next = StartingLocation; </pre> <p>There are two differences. First, the board no longer hard-codes the size of the board. Second, the board does not have to create BoardLocation objects of different kinds. If we create new subclasses of BoardLocation, we need to update the BoardLocationReader.</p> <p>If we want to extend the board, we simply need to change the text file and run the program.</p>

### File: Game.cs

#	Description
2	In previous versions of Game.cs we were able to just initialize the board. Now we are doing complex initialization, which is not supported at this line. Instead, we will initialize the myBoard attribute in the constructor.

7	Create an instance of a board location reader. The Game creates this so the Board is not bothered with knowing how to do it. This may seem trivial but if we later define a new kind of BoardLocationReader, we can give that to the Board instead and Board's code will remain the same.
8	Construct the board and pass in the BoardLocationReader. We have to do this in the Constructor as C# only allows simple initialization expressions above where we define the attributes.
9	Release the resources held by the BoardLocationReader. This is strictly not necessary since the object will be garbage collected and clean itself up. However, releasing resources as soon as we can is a good habit. Given that C# provides garbage collection, the primary kind of "memory leak" that you can still experiencing is forgetting to clean up resources or waiting to clean them up.

## Build & Execute Instructions

Follow these instructions to see the change to your system:

1. Edit Board.cs. Update the constructor.
2. Edit Game.cs. Change the initialization of myBoard and update the constructor.
3. (Optional)Update the MainApp.cs to remove the code that tests BoardLocationReader.
4. Edit Game.cs. Change the initialization of myBoard and update the constructor.
5. Compile using the standard csc \*.cs and execute with MainApp.

## Iteration 2b: Replacing Output with Events

This section supports no new functional requirements; we simply change how the system produces output. We introduce events into our system and move the output from our current classes to a new class, called an event receiver.

Events are a way for one part of a system to indicate that something has happened while another part of the system decides what to do in response. The event solution in .NET follows the GoF Observer pattern, see [10]. Some terminology will help:

Event Sender	A class that produces events. Any class can produce events; it simply needs to have an event attribute or property it exposes to clients. When events occur, the event sender must broadcast those events. This is referred to as raising or triggering an event.
Event Receiver	A class that registers interest with an event sender. An event receiver has a method that matches a signature that matches a delegate (which is something we are about to see). An event receiver handles or consumes events.
Wiring	Associating an event receiver with an event sender. This might also be called registering interest.

### A Problem

Systems have several logical layers, or high level areas of concern. A typical list might be:

- Presentation – A user interface
- Application – The flow through a system, which might be different for different users
- Business – The core logic of the system
- Integration – Connectivity to outside services, systems and databases

In addition, systems tend to have common data shared across the integration, business and application layers. For further details, please see “Appendix: Summary of the 5 + 1 Layers Pattern” starting on page 134.

In our system, we have mixed the presentation, application and business layers together. Our presentation layer is currently a text interface. We produce that output in our Player and BoardLocation classes. If we wanted to change the presentation, we have to change these classes. This might seem like normal programming, and it probably is, but it is not a good separation of concerns. It should be easy to change the user interface. A system that does not separate these concerns is not open to new clients in the future unless the new clients use the embedded user interface. This happens frequently when a new system will work with legacy systems. When the legacy systems are closed, often the only option is to use some kind of terminal emulation and simulate using the system as if a human operator were typing on the keyboard. There is a term for this, it is called screen scraping and while it can work, it is not generally elegant, easy or even very reliable.

We are moving towards a better separation of responsibility between the business logic (the rules of the game) and the presentation. To do this, instead of having our Player and BoardLocation classes actually produce the output, they will broadcast events to indicate something has happened. We will then write another class that is responsible for turning those events into the textual interface we have already experienced.

Before we do, we will work through a standalone example because this is complex enough to warrant going through it a few times.

### An Event Standalone Example

There are six parts to event handling:

1. Define a class that holds information about the event. This class must inherit from the C# class EventArgs. If the event is called Move, then we will have a class called MoveEventArgs. If an event is called StateChanged, then we will have a StateChangedEventArgs class. These classes are only required if we plan to send information along when the event occurs.
2. Define a Delegate which will allow wiring an event receiver to an event sender.
3. Define an event sender; a class that triggers events.
  - Add an event attribute or property to that class, using the Delegate we previously defined.
4. Create an event receiver class.
5. Wire the event receiver to the event sender.
6. Trigger events from within the event sender class.

**File: EventExample.cs**

```
1| using System;
2| // Step 1, define new event args class
3| public class StateChangedEventArgs : EventArgs {
4|     public string myDescription;
5|     public StateChangedEventArgs(string description) {
6|         myDescription = description;
7|     } // end StateChangedEventArgs constructor
8|     public string Description {
9|         get { return myDescription; }
10|     } // end Description property
11| } // end step 1, define new event args class
12|
13| // Step 2, define a delegate
14| public delegate
15|     void StateChangedEventHandler(object sender, StateChangedEventArgs e);
16| // end step 2, define a delegate
17|
18| // Step 3, define class that triggers events
19| public class ExampleEventGenerator {
20|     public event StateChangedEventHandler StateChangedListeners;
21|     protected virtual void OnStateChanged(StateChangedEventArgs e) {
22|         if (StateChangedListeners != null) {
23|             // Step 6, trigger events
24|             StateChangedListeners(this, e);
25|         } // end if
26|     } // end OnStateChanged method
27|     public void Run() {
28|         for(int i = 0; i < 10; ++i) {
29|             StateChangedEventArgs e = new StateChangedEventArgs("" + i);
30|             // Step 6, call a method to trigger events
31|             OnStateChanged(e);
32|         } // end for loop
33|     } // end Run method
34| } // end step 3, define class that triggers events
35|
36| // Step 4, define event receiver
37| public class TextUI {
38|     private string myName;
39|
40|     public TextUI(string name) {
41|         myName = name;
42|     } // end TextUI constructor
43|     public void
44|         DisplayOngoingResults(object sender, StateChangedEventArgs e) {
45|         Console.WriteLine
46|             ("State Change: {0} - {1}", myName, e.Description);
47|     } // end DisplayOngoingResults method
48| } // end step 4, define event receiver
49|
```

```
50| public class MainApp {  
51|     public static void Main() {  
52|         TextUI ui1 = new TextUI("UI 1");  
53|         TextUI ui2 = new TextUI("UI 2");  
54|         ExampleEventGenerator eg = new ExampleEventGenerator();  
55|  
56|         // Step 5, wire event receiver to event sender  
57|         eg.StateChangedListeners +=  
58|             new StateChangedEventHandler(ui1.DisplayOngoingResults);  
59|         eg.StateChangedListeners +=  
60|             new StateChangedEventHandler(ui2.DisplayOngoingResults);  
61|         // End step 5, wire event receiver to event sender  
62|  
63|         eg.Run(); // begin triggering events.  
64|     } // end Main method  
65| } // end MainApp class
```

## Code Explained

This is our first example of multiple classes in the same file. We do this for simplicity. This is a practice example and we do not intend this to become part of any system.

### 1. Define an EventArgs class

#	Description
3	Define a class to hold information about an event. The name of the class should, by convention, end in EventArgs. The class must inherit from the EventArgs class.
4 – 11	The remainder of this class contains any information that should go along with an event when it occurs. The constructor accepts this information and records it in attributes. We might have properties that allow access to that information as well.

### 2. Define a delegate

#	Description
14 – 15	The delegate keyword introduces a special kind of class. Underneath the covers, C# creates a class by the same name as the delegate. It has a constructor that takes as a parameter, a method with the same signature as the delegate. In our example, any method that returns void and takes 2 parameters, an object and a StateChangedEventArgs class, will suffice.  Delegates are C#'s way of handling pointers to functions in a safe way. C# calls this a delegate since a delegate is able to stand in for something else. In this case, the delegate stands in for a call to another function.  We create instances of the delegate on lines 53 and 55.

### 3. Define an event sender

#	Description
19	Define a class. Looking at this part of the class definition, there is no way to tell that this class triggers events.
20	Define an event attribute. This is the thing that clients will use to register an interest in events generated by instances of this class. This is public for accessibility. When using the event keyword, the next thing after event must be a delegate type. We register event listeners on lines 52 and 54 using +=.
21	Define the method that will actually broadcast events. This method is virtual so that subclasses can override it. This method is protected so that subclasses can access it but not outsiders. Its name starts with “On” and ends with the kind of event, StateChanged in this case. It takes one parameter, the EventArgs class we already defined.
22	This is an idiom. Check for null. If the comparison to null returns true, there are no listeners, do

– 25	not try to broadcast events. If there are listeners, broadcast to all of them. The first parameter is the current object, the sender of the event, the second is the EventArgs instance passed in as the parameter.
27 – 33	Define a simple method to trigger events. This method simply fires 10 events. It creates an instance of StateChangedEventArgs, passing in a string representation of an integer. It then calls its own internally-defined OnStateChanged method to broadcast this event.

#### 4. Create an event receiver

#	Description
37	Define a class that listens to events. Looking at this part of the definition there is no way to tell this class can handle events. What makes it a candidate for events is that it has a method whose signature matches the delegate defined earlier.
38	Give each instance a name so we can distinguish between multiple listeners.
43 – 47	Define a method that can handle events. The thing that makes this method a candidate delegate is that its signature matches the delegate defined above on line 13. Since this is an instance method (no static modifier), we will have to create an object of the TextUI class to be able to create a delegate that points to this method. We do this on lines 50 and 52.  Our response to an event occurring is to simply display a message showing the name of the particular TextUI object and the description provided in the StateChangedEventArgs.

#### 5. Wire the event receiver to event sender

#	Description
50	Define our MainApp class to create the necessary objects and wire everything together.
52	Create an object that is suitable as an event receiver. Give it the name “UI 1”.
53	Create a second object that is suitable as an event handler. It has the name “UI 2” so we can distinguish its output from UI 1’s output.
54	Create the object that will trigger events. It starts with no event listeners.
57 – 58	First create an instance of StateChangedEventHandler. This is the delegate we defined on lines 12 and 13 above. To create the delegate we provide a handle to a method that has the correct signature. To get a handle to the method, we use object.MethodName. Once we have created the delegate, we append it to the list of event handlers by using +=.
59 – 60	We do the same thing but with a different instance of the TextUI class. This means that there are two objects interested in the events triggered by the ExampleEventGenerator instance. It happens that the first registered event listener will get the event first. Since we have two instances of the same class, and both do the same thing, we will see the output repeated. We can tell the differences because we provided different names to the two TextUI classes.

#### 6. Trigger Events

#	Description
63	Call the method we wrote to trigger events. This method simply generates several events.

#### Build & Execute Instructions

Follow these instructions to see the change to your system:

1. Create a separate directory for this example, for example c:\C#\Event.
2. Create the EventExample.cs class.
3. Compile the EventExample.cs class using csc \*.cs
4. Execute the example, you will see the following output:



```
c:\C#\Event>EventExample
State Change: UI 1 - 0
State Change: UI 2 - 0
State Change: UI 1 - 1
State Change: UI 2 - 1
State Change: UI 1 - 2
State Change: UI 2 - 2
State Change: UI 1 - 3
State Change: UI 2 - 3
State Change: UI 1 - 4
State Change: UI 2 - 4
State Change: UI 1 - 5
State Change: UI 2 - 5
State Change: UI 1 - 6
State Change: UI 2 - 6
State Change: UI 1 - 7
State Change: UI 2 - 7
State Change: UI 1 - 8
State Change: UI 2 - 8
State Change: UI 1 - 9
State Change: UI 2 - 9
```

## A Solution

Following the steps described in “An Event Standalone Example” starting on page 61, we have several things we will need to do. Before we begin, we need to select a name for this new kind of event. Things happen in the game during a turn so we will call our event a “Turn”.

### Step 1 & 2: TurnEventArgs Class and Delegate

We will need to create a class called TurnEventArgs. We also need to define a delegate. Following the naming convention, the delegate will be named TurnEventHandler.

#### File: TurnEventArgs.cs

```
1| using System;
2| namespace BoardGame {
3|
4|     public enum TurnSteps { START, CONT, PASS, LAND, FINISH }
5|
6|     public class TurnEventArgs : EventArgs { // Step 1
7|         private string myDescription;
8|         private TurnSteps myAction;
9|
10|         public TurnEventArgs(string description, TurnSteps action) {
11|             Assert.NotNull(description, "description");
12|
13|             myDescription = description;
14|             myAction = action;
15|         } // end TurnEventArgs constructor
16|
17|         public string Description {
18|             get { return myDescription; }
19|         } // end Description property
20|
21|         public TurnSteps Action {
22|             get { return myAction; }
23|         } // end Action property
24|     } // end TurnEventArgs class
25|
26|     // step 2
27|     public delegate void TurnEventHandler(object sender, TurnEventArgs e);
28| } // end BoardGame namespace
```

#	Description
4	Use an enumeration to define a set of constants. These constants identify parts of a turn. Event

	senders will use these constants to identify the part of the turn from which the event emanates. Event receivers will use these constants to determine how to format output.
6	Create a class. Name it TurnEventArgs, following the naming convention. This class must inherit from the EventArgs class.
7 – 8	Define two attributes. This information will be available to the event receiver. We are converting our output to using events, so our event receiver will print the description and use the action to know to which part of a step the TurnEventArgs relates.
10	Standard constructor. Store the provided parameters in attributes.
12	Notice that we do not validate the action parameter. It is an enumeration. It cannot be null. It could be outside of the range of valid values but only if someone uses a cast. We could check to make sure that the provided value is one of the enumeration values, we chose not to.
17	Standard Property.
21	Standard Property.
27	Define a delegate for event handling. All event delegates take two parameters. The first is object. The second is some either EventArgs or a subclass of event args. We created TurnEventArgs because we want to send specific information with our events.

### Step 3 & 6: Define an Event Sender & Generate Events

An event sender is any object that has an event as an attribute or property. In our system as it is now, output comes from the Player and various BoardLocation classes. This means we need to change those classes to trigger events instead of issuing output.

For each of the following files we only show the added or changed material. The other methods already defined remain the same.

#### Updates: Player.cs

```

1| // Add an event attribute
2| public event TurnEventHandler TurnListeners; // step 3
3|
4| // Update TakeATurn
5| public void TakeATurn(PairOfDice dice) { // step 6, all calls to OnTurn
6|     OnTurn(Name + " on " + Location + " has " + Cash, TurnSteps.START );
7|     dice.Roll();
8|     OnTurn("rolls " + dice.Value, TurnSteps.CONT);
9|     Location = Location.Next;
10|     for(int i = 1; i < dice.Value; ++i) {
11|         Location.PassOver(this);
12|         Location = Location.Next;
13|     } // end for loop
14|     OnTurn("will land on " + Location, TurnSteps.CONT);
15|     Location.LandOn(this);
16|     OnTurn("and now has " + Cash, TurnSteps.FINISH);
17| } // end TakeATurn method
18|
19| // Add a support method for triggering events
20| private void OnTurn(string description, TurnSteps action) {
21|     OnTurn(new TurnEventArgs(description, action));
22| } // end OnTurn method
23|
24| // Add the standard method for triggering events
25| protected virtual void OnTurn(TurnEventArgs e) {
26|     if(TurnListeners != null) {
27|         TurnListeners(this, e);
28|     } // end if
29| } // end OnTurn, note this method is overloaded with above OnTurn method

```

#	Description
2	Define an event attribute called TurnListeners. This allows event receives to be wired to

	instances of the class.
6	Use a private method, OnTurn, to send an event with two data elements. The first is a string with the name of the player, their location and current cash amount. The second is a constant, START, indicating that this is the beginning of a turn.
8	Use a private method to generate another event. This time it contains the roll value and the constant CONT, indicating this is a continuation of the turn.
14	Use a private method to generate an event that contains information about where the Player is about to land. This is also a continuation of the turn.
16	Use a private method to generate another event. This final event uses the FINISH constant, indicating it is the last thing the turn for this player will generate for an event.
20 – 22	This is a support method to assist in generating an event. When we send events, we must provide an instance of an EventArgs class. In our case we are using the TurnEventArgs class, which requires two parameters. Rather than have several “new TurnEventArgs(…)” in the TakeATurn method, we have a simple support method. It simply creates an instance of the TurnEventArgs with the provided parameters and calls OnTurn, which actually broadcasts the event.  Note that this and the next method are overloaded. See the next paragraph.
25 – 29	This method follows an idiom. It is protected and virtual to allow future subclasses override it. It checks to see if the TurnListeners is null. Assuming it is not, it broadcasts the events. To do this, it uses the parenthesis, ( ), on the TurnListeners attribute. The first parameter is the current object; the second is the TurnEventArgs class.  This method is overloaded with the first OnTurn method. Overloading allows two methods to have the same name but take different parameters.

**Updates: BoardLocation.cs**

```

1| // Add an event attribute
2| public event TurnEventHandler TurnListeners;
3|
4| // Add a support method to trigger events
5| protected virtual void OnTurn(string s, TurnSteps action) {
6|     OnTurn(new TurnEventArgs(s, action));
7| } // end OnTurn method
8|
9| // Add the standard method for triggering events
10| protected virtual void OnTurn(TurnEventArgs e) {
11|     if (TurnListeners != null) {
12|         TurnListeners(this, e);
13|     } // end if
14| } // end OnTurn, note this method is overloaded with above OnTurn method

```

#	Description
2	This follows the form we have seen so far. Define an event attribute.
5	Define a support method that subclasses can use or override. This method does exactly the same thing that the OnTurn method did in the Player class.
10	This is the actual method that broadcasts events. Notice that we could copy this directly from the Player class.

**Updates: SpringBoard.cs**

```

1| // Update the LandOn method
2| public override void LandOn(Player p) {
3|     BoardLocation dest = CalculateDestination();
4|     p.Location = dest;
5|     passCount = 0;
6|     OnTurn(p.Name + " springing to " + dest.Name, TurnSteps.LAND);
7| } // end LandOn method
8|

```

```

9| // Update the PassOver method
10| public override void PassOver(Player p) {
11|     if (passCount < 13) {
12|         ++passCount;
13|     } // end if
14|     OnTurn(p.Name + " incremented pass count", TurnSteps.PASS);
15| } // end PassOver method

```

#	Description
6	Use the OnTurn method inherited from BoardLocation to trigger an event. Since this is in the LandOn method, we use the LAND constant.
14	Same comment as line 8. This is during a PassOver method so we use the PASS constant.

#### Updates: StartingLocation.cs

```

1| // Update the LandOn method
2| public override void LandOn(Player p) {
3|     p.Cash += SALARY;
4|     OnTurn(p.Name + " receives " + SALARY, TurnSteps.LAND);
5| } // end LandOn method
6|
7| // Update the PassOver method
8| public override void PassOver(Player p) {
9|     p.Cash += SALARY;
10|    OnTurn(p.Name + " receives " + SALARY, TurnSteps.PASS);
11| } // end PassOver method

```

#	Description
2, 8	Other than the contents of the string, this looks the same as what we did in SpringBoard.cs. It also looks similar to Player.TakeATurn().

#### Update: SurpriseBill.cs

```

1| // Update the LandOn method
2| public override void LandOn(Player p) {
3|     p.Cash -= fine;
4|     OnTurn(p.Name + " pays " + fine, TurnSteps.LAND);
5| } // end LandOn method

```

#	Description
2	Now that we have the hang of this, it is quite easy to generate events. The base class provides a good deal of assistance.

### Step 4: Create an Event Receiver

Next, we need to create a class that can handle these events and generate output. We created a simple class that will mimic the output we were already generating:

#### File: TextUI.cs

```

1| using System;
2| namespace BoardGame {
3|     public class TextUI {
4|         public void DisplayTurnEvent(object sender, TurnEventArgs e) {
5|             switch(e.Action) {
6|                 case TurnSteps.START:
7|                     Console.WriteLine(e.Description);
8|                     break;
9|                 case TurnSteps.CONT:
10|                     Console.WriteLine("\t{0}", e.Description);
11|                     break;
12|                 case TurnSteps.FINISH:
13|                     Console.WriteLine("\t{0}", e.Description);
14|                     break;
15|                 case TurnSteps.PASS:
16|                     Console.WriteLine("\tPassing: {0}", e.Description);

```

```

17|         break;
18|         case TurnSteps.LAND:
19|             Console.WriteLine("\tLanding: {0}", e.Description);
20|             break;
21|     } // end switch
22| } // end DisplayTurnEvent method
23| } // end TextUI class
24| } // end BoardGame namespace

```

#	Description
2	Nothing about this line indicates that it is or is not an event receiver. The thing that makes it a candidate is that it contains a method with a signature that matches the one defined by our delegate.
4	This is the method that matches the signature of the delegate. This method is an instance method (meaning it does not have the static modifier). To call this method or to create a delegate of this method, we must have an object available. To create a delegate of this method, we must provide the delegate with both the instance and the method. To do this, assuming we have an instance called “viewer”, we simply pass in viewer.DisplayTurnEvent.
5 – 21	What this event listener does with the events is its own business; event sources do not have any control over what a listener does with events. In this case we use the Action property of the TurnEventArgs class to determine how many tabs to put into the output. We also use the Action property to add some information to the output such as “Landing:”.  The implementation of this class is somewhat arbitrary. It looks like it does to make the output from the previous version look close to this version of the output.

## Step 5: Wire Event Listeners

Instance registration happens in two places. It starts in MainApp.cs where we create the event listeners and pass them into the Game constructor. It continues in the Game class when the Game class explicitly adds the event listener to the Player and BoardLocation objects.

### File: MainApp.cs

```

1| public class MainApp {
2|     public static void Main(string[] args) {
3|         // contents removed
4|         TextUI t = new TextUI();
5|         TurnEventHandler h = new TurnEventHandler(t.DisplayTurnEvent);
6|         Game g = new Game(playerCount, h);
7|         g.Play(roundCount);
8|     } // end Main method
9| } // end MainApp class

```

#	Description
4	We want an instance of the TextUI responsible for output. We are going to register a method of a TextUI as the event handler. The method is an instance method, which can only be called with an object. When we create the event handler it will be for a particular method and for this particular object.
5	Create a delegate. The type TurnEventHandler is a delegate. To create a delegate, provide a reference to a method. Since the method is an instance method, we use an existing object. The expression v.DisplayTurnEvent gives the handler, h, 2 things. First, v, second, DisplayTurnEvent. When the event generator uses this delegate, the net effect is the message DisplayTurnEvent will be sent to the object referenced by v.
6	Create the game, passing in the handler. We give the game the responsibility of hooking the event handler with all of the event generators.

### Updates: Game.cs

```

1| // Update the Game constructor
2| public Game(int playerCount, TurnEventHandler handler) {

```

```

3|     Assert.NotNull(handler, "handler");
4|     // the middle remains the same
5|     RegisterHandler(handler);
6| } // end Game constructor
7|
8| // Add a private support method
9| private void RegisterHandler(TurnEventHandler h) {
10|     foreach(Player p in players) {
11|         p.TurnListeners += h;
12|     } // end foreach loop
13|     BoardLocation current = myBoard.StartingLocation;
14|     do {
15|         current.TurnListeners += h;
16|         current = current.Next;
17|     } while(current != myBoard.StartingLocation); // end do loop
18| } // end RegisterHandler method

```

#	Description
5	Call a method to wire the event receiver.
10 – 12	Iterate over all of the players. Wire the event receiver with each player. The way to wire an event handler is to use an attribute or property with an event modifier. Using += will append the receiver to the list of current receivers.
13 – 17	Same comment as above. Each of the BoardLocation objects might also generate output. Note that we cannot use foreach to iterate over the board locations. The foreach keyword requires either an array or a collection. A collection has to meet certain criteria such as having a GetEnumerator method, which returns an enumerator. An enumerator must have a Current property and a MoveNext method. (The BoardLocationReader is very close to an enumerator, it does not generate all the necessary exceptions.)

## Build & Execute Instructions

1. Create the TurnEventArgs.cs file and compile the system. Remove any errors before moving on.
2. Update Player.cs. Compile and remove any errors.
3. Update Game.cs and MainApp.cs.
4. Compile and execute. You should see output.
5. Update all of the BoardLocation hierarchy.
6. Compile and execute. You should see output similar to the following:

```

c:\C#\BoardGame>MainApp
Player#0 on StartingLocation(Starting Location) has 1500
    rolls 5
    will land on BoardLocation(Bus)
    and now has 1500
Player#1 on StartingLocation(Starting Location) has 1500
    rolls 5
    will land on BoardLocation(Bus)
    and now has 1500
Player#0 on BoardLocation(Bus) has 1500
    rolls 9
    will land on BoardLocation(Scottsdale)
    and now has 1500
Player#1 on BoardLocation(Bus) has 1500
    rolls 10
    will land on BoardLocation(Rail)
    and now has 1500

```

## Summary

This section covers events. Events are used for graphical user interfaces and throughout the .NET framework. An event allows one object in a system to generate information that another object cares

about. Furthermore, the onus of deciding what to do with that information is not placed on the generator but the receiver.

In our system we have managed to remove output from our Player, and BoardLocation classes. Those classes must still generate information, but they are not responsible with what to do with in. We created one simple kind of event handler, TextUI that mimics the output from the previous iteration. We could just have easily created a graphical user interface that uses those same events to draw the player moving around the board. We would have to add event generation to the BoardLocation's implementation of LandOn and PassOver but this is easy to do.

There are several steps for generating events. To review them, look at "A Problem" starting on page 61. We have used events to separate something that might at first seem subtle. Our BoardLocation and Player classes know when something happened that was worthy of note. Taking a turn, landing on some locations, passing over others, etc. In previous iterations they also were responsible for knowing what to do when something happened. We did not distinguish between these two ideas: something happened, inform the user. We have made this distinction by using events. The Player and BoardLocation objects let the world know that something happened. The TextUI decides what to do when that happens. Our first attempt did not make the distinction so there was no way to offer flexibility. By separating these concerns we can now vary each independently. The point where these two things come together is the event.

## Iteration 3: Abstract Classes

### Introduction

In this chapter we up the ante just a bit. We tackle quite a few functional requirements. Given the size and complexity of the requirements we will use a series of seven builds. Each of these builds could almost be a chapter. We also transition from giving code before design to spending a bit more time in analysis, then design before moving into code. We also bring you more into the design process and start to justify our designs a bit more than we have.

We recommend you approach this chapter build by build or may be a few builds at a time. By the end of this chapter you will look back and realize

- That you have a pretty good grasp of reading UML diagrams,
- You have a better idea of how to approach a problem a bit more formally,
- You will have worked through an example of using small steps to satisfy requirements,
- You will have started using more formal layer separation in preparation of supporting multiple user interfaces on top of the same model.

This is quite a bit of material to cover, but remember that mastery is not the goal. Awareness with concrete examples is what we're after.

### New Requirements

In “Iteration 2a: Reading from a File” starting on page 51 we saw several new types of BoardLocation classes we have yet to handle in our system. These types are city, random, transportation, bnb, and utility. In this section we will provide the requirements for all of these new types. This iteration will only look at some of the requirements for three of these remaining classes. The three we look at will lead us to abstract classes, abstract properties and sealing features.

### City

Players can own cities. If a player owns a city, then other players must pay a city tax to stay there. This fine is assessed when a player arrives on the city. A player must pay the city tax every time they land on the location, sort of like the fee people pay to drive a car in to London from July 2003.

Initially all cities are Unowned. When a player lands on a city that is not owned, the player has the option of buying the city. If the player does not, nothing happens. If the player does buy the city, then all subsequent players who land on that city must pay the city tax. The cost to purchase a city is the third entry on each city line in the BoardLocation.txt file. The basic tax for each city is the fourth entry on each line.

All cities belong to a state. If a player owns all of the cities in the same state, the city tax automatically doubles. The larger the government, the more it costs for the same thing. The state the city belongs to is the fifth entry on the line describing the city.

Once a player owns a city, they can plan special events for the city. A special event, such as the Olympics, costs some amount and has some duration. If a player plans an event and pays for it, all Players must pay a new tax amount based on the cost of the event. A Player passing through a city with a special event might be slowed down by the special event. Whether or not that happens depends on the event. A player may choose to plan a special event only during her turn and before she moves. This disallows a player from purchasing a RealEstate object and immediately planning a special event.

A player may raise the state tax if they own all of the cities in a state. Luckily state tax is still regionalized so they must pay to raise the tax on a city by city basis. Each city may have its tax raised



up to five times, but no two cities can vary more than one raise in taxes. The price to raise taxes once is 20% of the original purchase price, rounded down. The increase in taxes is 5% over the current tax amount, rounded down. A player may choose to raise taxes only during her turn and before she moves. This disallows a player from purchasing a RealEstate object, forming a set of all the cities in a group and raising taxes, all during the same turn. A player may optionally sell back tax increases. If they do they get back half of their investment. She can only do this at the beginning of her turn.

Someone who owns a city may take a lien out against the City. The lien amount may be for up to two times the original purchase price of the City. The player may only do this during her turn, either before moving or in response to having to pay some amount of money. When the City has a lien, other Players do not pay taxes for landing on the city. The player can still schedule special events but there are no taxes collected. The lien may be paid back only during the Player's turn and only before she moves. The lien amount increases by 8% of its current value, rounded down, each time the Player begins another turn. When the current lien amount is greater than double the original lien amount, the Player must pay back the entire lien. If she is not able to, then she must pay back all the Cash she has and the City becomes available for purchase for whoever lands on it first. If this happens and the Player had increased state taxes, the Player must remove the state taxes on all of the cities. They get back half of their investment.

### **Paying fines**

A player may be assessed a fine. A fine is any amount a player must pay. Things that are not fines include:

- Landing on an Unowned RealEstate and having the option to purchase it.
- Being forced to pay back a Lien on a RealEstate object.

When the player must pay a fine, the money must come from the following sources, listed in order of where the money comes from first:

- Cash,
- City taxes,
- Taking out a Lien on a property,

For example, if a player must pay 500, they first pay out of their Cash. If that is enough to cover the fine, then simply deduct the amount. If that is not enough, then the player must begin to sell off city tax increases. If that is not enough, then the player must begin to take out Liens. If the player is unable to pay the fine, they are out of the game. All of their RealEstate objects revert to Unowned.

### **Transportation**

Players can own the various transportation locations. The purchase rules, lien and Event rules are the same for transportation as they are for a city, with the exception that it is not possible to increase transportation taxes. (Remember this is a simulation and not the real thing.)

If a player owns one transportation location, they receive 50 when other players land on that transportation location. If a player owns two, they receive 100. Three owned by the same player will cost 200 and all four will cost 400.

The third entry on each line describing a transportation location contains the cost to purchase the transportation object.

### **Utility**

Players can own utilities. The purchase rules for transportation and lien rules are the same for Utility as they are for Transportation.

The cost for landing on a utility is equal to the current value shown on the PairOfDice times some number. If only one utility is owned by any player, the cost is 15 times the current value shown. If

both utilities are owned, regardless if it is by the same player or not, the cost is 40 times the current value shown on the PairOfDice.

The third entry on each line describing a utility location contains the cost to purchase the utility.

## Bnb

A bed and breakfast always costs to land on it. It is not owned by other players, but other players can opt to stay on a bed and breakfast for up to five turns, paying the cost of the first night plus the total paid so far. For example, if the cost is 25, the first turn costs 25. The second costs 25 (base) plus the total paid so far, 25, for a total of 50. The next night is 75, then 100, then 125. This might sound expensive, but landing on cities can often be much more expensive.

A player must opt to stay before attempting to move. If they do pay, they do not roll the dice or move.

## Random

A random location does one of a number of things to a player. The list of things is:

- Nothing
- Move to a particular city
- Move to the next transportation location
- Move to a random location on the board
- Receive some fixed amount of money
- Pay some fixed amount of money to the ether
- Pay a fixed amount of money to each player
- Pay a fixed amount of money times the number of events the player has scheduled
- Pay a fixed amount of money times the number of state taxes the player has bought
- Receive a free stay at the location of the player's choice (the player who should have received the cash still does so, it just comes from the ether)
- Take another turn
- Go directly to the starting location

In all cases where a player moves, the effect is as if the player moves to that location, passing each location along the way and ultimately landing on the destination location. For example, if a player lands on an un-owned transportation, they can purchase it.

## Build 1: City only

For this iteration we are only going to handle City, Transportation and Utility. Before we handle all three, however, we are going to only partially take care of City. After we have created City, we will look at transportation and Utility and see what is common between the three classes. After that we will be able to justify using an abstract class, an abstract property and a sealed method.

### File: City.cs

```
1| namespace BoardGame {
2|     public class City : BoardLocation {
3|         private int price;
4|         private Player owner;
5|         private bool hasLien;
6|         private int baseTax;
7|
8|         public City(string name, int price, int baseTax)
9|             : base(name) {
10|             Assert.Argument(price > 0, "price", price, "should be > 0");
```

```

11|         Assert.Argument(baseTax > 0, " baseTax ", baseTax, "should be > 0");
12|
13|         this.price = price;
14|         this.baseTax = baseTax;
15|     } // end City constructor
16|
17|     public int Price {
18|         get { return price; }
19|     } // end Price property
20|
21|     public bool IsOwned {
22|         get { return Owner != null; }
23|     } // end IsOwned property
24|
25|     public Player Owner {
26|         get { return Owner; }
27|     } // end Owner property
28|
29|     public bool IsOwnedBy(Player p) {
30|         return p == Owner;
31|     } // end IsOwnedBy method
32|
33|     public int Tax {
34|         get { return baseTax; }
35|     } // end Tax property
36|
37|     public bool HasLien {
38|         get { return hasLien; }
39|         set { hasLien = value; }
40|     } // end HasLien property
41|
42|     public override void LandOn(Player p) {
43|         if(!IsOwned) {
44|             OfferMyselfForSaleTo(p);
45|         } else if(!HasLien) {
46|             CalculateAndChargeTaxFor(p);
47|         } else {
48|             OnTurn(p.Name + " avoided tax on " + Name, TurnSteps.LAND);
49|             // Nothing: No payments when there are Liens
50|         } // end if
51|     } // end LandOn method
52|
53|     private void OfferMyselfForSaleTo(Player p) {
54|         p.Cash -= price;
55|         owner = p;
56|         OnTurn(p.Name + " just bought " + Name, TurnSteps.LAND);
57|     } // end OfferMyselfForSaleTo method
58|
59|     private void CalculateAndChargeTaxFor(Player p) {
60|         if(!IsOwnedBy(p)) {
61|             int tax = Tax;
62|             p.Cash -= Tax;
63|             Owner.Cash += Tax;
64|             OnTurn(
65|                 p.Name + " just paid " + Owner.Name + " " + tax + " tax ",
66|                 TurnSteps.LAND
67|             ); // end call to OnTurn
68|         } // end if
69|     } // end CalculateAndChargeTaxFor method
70| } // end City class
71| } // end BoardGame namespace

```

#	Description
2	<b><i>public class City : BoardLocation {</i></b> Define a new class called City. It is a kind of BoardLocation object (for now).
3	Define several attributes. The Transportation and Utility classes also have price, owner and hasLien. BaseTax is unique to City. After this first build, we will move the common attributes
–	

#	Description
6	up to an intermediate class between BoardLocation and City.
8 – 15	Define a standard constructor. Pass the name parameter up to the base class constructor. Store all of the remaining parameters in to attributes.
16 – 40	Define several different properties. Notice that IsOwned, Owner and IsOwnedBy all seem to be doing similar things. IsOwned is a check. Rather than comparing the owner attribute to null to deduce that the City is not, we have a Property. This will make the code a little easier to understand.  The HasLien property is there so we can completely code the basic landing rules. We do not support all of the ways in which the tax can increase but we do support the three basic states of a location: Unowned, Owned, Mortgaged.
42	Something happens when we land on this location. Replace the inherited version.
43	Check to see if this location is owned.
44	The location is not owned, it will offer itself for sale to the current player.
45	The location is owned, does it have a lien?
46	No it does not have a lien. Determine and charge the tax the current player might pay.
47	The location is owned and it has a lien.
48	Trigger an event that states the player who just landed avoided paying taxes because of the lien.
53	A support method where the City sells itself to the current player. We are not asking the player or worrying about negative balances. If a player lands on the location, we simply sell the property to the current player. This is only Build 1 after all.
54	Deduct the price of the City from the player's balance.
55	Set the owner of the city to the current player.
56	Trigger an event that the current player just bought the City.
59	A support method to calculate taxes and charge the player.
60	Is the current player the owner of this location? If so, then nothing happens.
61	The current player is not the owner. Calculate the tax owed. Note that we are not performing full tax calculations. Right now we only charge base tax. Again, this is build 1.
62	Make the current player pay the tax.
63	Give the owner the cash just paid by the current player.
64	Trigger an event describing the taxes paid.

### File: BoardLocationReader.cs

The only change to BoardLocationReader is to update the switch statement in SetCurrentFrom:

```

1|  case "city":
2|      int price = Int32.Parse(tokens[2]);
3|      int baseTax = Int32.Parse(tokens[3]);
4|      current = new City(name, price, baseTax);
5|      break;
```

#	Description
1	<b>case "city":</b> This is the only case we need to change right now. As we introduce the other classes we will change other cases as well.
2	<b>int price = Int32.Parse(tokens[2]);</b> Define a temporary variable called price. Convert the token at index 2 (third entry on the line) into an integer. If the file is not well formed, this line will throw a System.FormatException. Warning: When we support the Utility and Transportation, we will need to get the price of

#	Description
	each kind of object from the BoardLocations.txt file. We will use the local variable price, currently on line 2 in the above example. We will need to move the definition of this variable outside of the switch statement to make everything compile.
3	<code>int baseTax = Int32.Parse(tokens[3]);</code> The token at index 3 represents the basic tax the current player pays to the owner when landing on the location.
4	<code>current = new City(name, price, baseTax);</code> Construct a new City object and set the current attribute.

## Build & Execute Instructions

1. Create the file City.cs.
2. Update BoardLocationReader.cs by changing the “city” case in the method SetCurrentFrom.
3. Compile and execute:

```
c:\C#\BoardGame>csc *.cs
c:\C#\BoardGame>MainApp 2 100
Player#0 on StartingLocation(Starting Location) has 1500
    rolls 5
    will land on BoardLocation(Bus)
    and now has 1500
Player#1 on StartingLocation(Starting Location) has 1500
    rolls 11
    will land on City(Scottsdale)
    Landing: Player#1 just bought Scottsdale
    and now has 1360
Player#0 on BoardLocation(Bus) has 1500
    rolls 4
    will land on City(Austin)
    Landing: Player#0 just bought Austin
    and now has 1380
Player#1 on City(Scottsdale) has 1360
    rolls 6
    will land on BoardLocation(Random Effect)
    and now has 1360
// output cropped
Player#0 on BoardLocation(Natural Gas) has 925
    rolls 7
    will land on City(Baton Rouge)
    Landing: Player#0 just bought Baton Rouge
    and now has 725
Player#1 on City(Alexandria) has 90
    rolls 11
    Passing: Player#1 incremented pass count
    will land on City(Redmond)
    Landing: Player#1 just paid Player#0 22 tax
    and now has 68
```

## Observations

Since we already have a base class in place and a single place where we map lines in a text file into instances of different classes, adding this class involved three steps:

1. Create the class as a subclass of BoardLocation.
2. Override one or more virtual methods (to get different behavior).
3. Update BoardLocationReader.cs

Now that we have done this for City, we will look at adding both Transportation and Utility.

## Build 2: Transportation and Utility

Before going too far with Transportation and Utility, first review what we have already done and see what applies. Here is `City.LandOn(Player p)`:

```

1| public override void LandOn(Player p) {
2|     if(!IsOwned) {
3|         OfferMyselfForSaleTo(p);
4|     } else if(!HasLien) {
5|         CalculateAndChargeTaxFor(p);
6|     } else {
7|         OnTurn(p.Name+ " avoided tax on " + Name, TurnSteps.LAND);
8|         // Nothing: No payments when there are Liens
9|     } // end if
10| } // end LandOn

```

Here is a table with the steps and what we do for Transportation and Utility:

	Transportation	Utility
2. <code>if(!IsOwned)</code>	Offer itself for sale to the current player. Same as City.	Offer itself for sale to the current player. Same as City.
4. <code>} else if(!HasLien) {</code>	Calculate and charge tax. Same as for city.	Calculate and charge tax. Same as for city.
6. <code>} else {</code>	Do nothing. Same as for city.	Do nothing. Same as for city.

In all three situations, these kinds of locations do the same thing. If that is the case, then are these the same or different classes? Where is there variance? There are several ways that these three classes vary, and some ways they are the same:

- The way we calculate tax is different for each of them. Each City object has a base tax. Each Transportation object has a base tax of 25. Utility objects use the current value shown on the pair of dice and multiply this value by some amount.
- When players start to own groupings, the tax changes. For Cities, the tax doubles when a player has all City objects within a state. For Transportation, the value doubles for each transportation object owned by the same player. For the utility, if only one is owned the roll multiplier is 15, if both are owned, regardless of by whom, the multiplier is 40.
- We can pay to increase the Tax on a City but not on Transportation or Utility.
- We can plan events for all three kinds of locations.
- All three potentially have an owner.

Next, consider the `CalculateAndChargeTaxFor(Player p)` method:

```

1| private void CalculateAndChargeTaxFor(Player p) {
2|     if(!IsOwnedBy(p)) {
3|         int tax = Tax;
4|         p.Cash -= Tax;
5|         Owner.Cash += Tax;
6|         OnTurn(
7|             p.Name + " just paid " + Owner.Name + " " + Tax + " tax ",
8|             TurnSteps.LAND
9|         ); // end call to OnTurn
10|     } // end if
11| } // end CalculateAndChargeTaxFor method

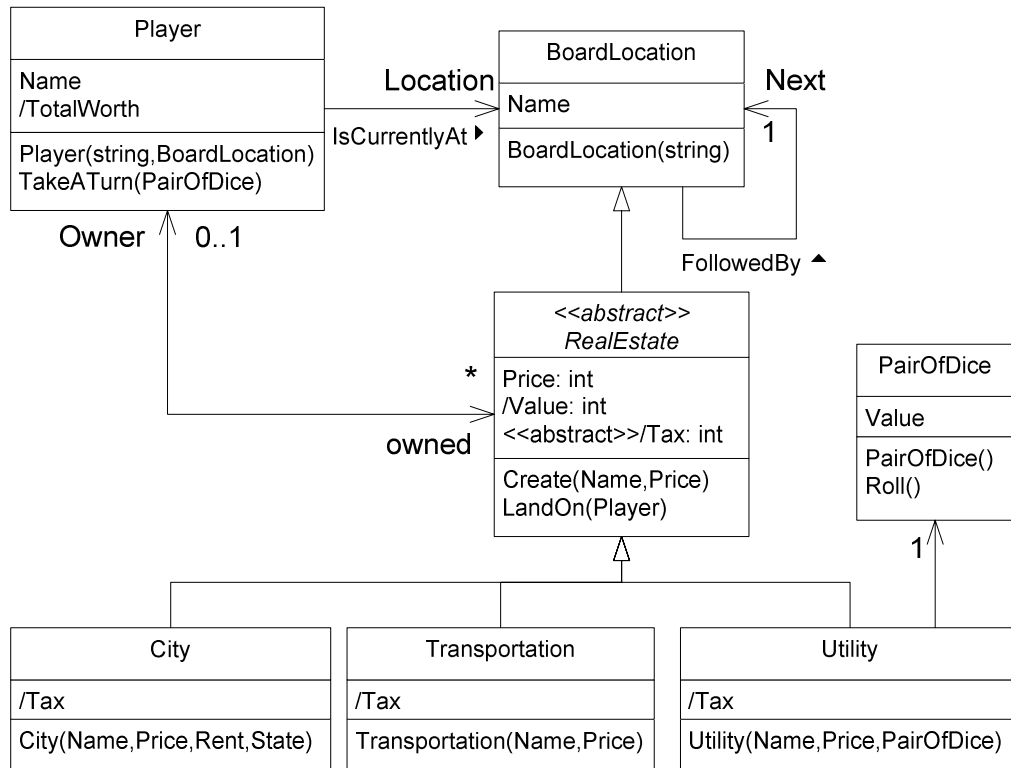
```

How does this method change to support the Transportation and Utility classes? Looking at the entire method, the only place where there needs to be variance is in the calculation of Taxes. We calculate taxes on line 3 in the above example.

There is enough commonality to warrant an intermediate base class. Assume we call this base class `RealEstate`. `RealEstate` should have all of the common elements including knowing the price and the owner, as well as whether it is owned, not-owned or has a lien against it.

We can enforce the rules for landing on the location by sealing the LandOn method, which we will see. We can also force subclasses to perform their own Tax calculations by making the property abstract in the RealEstate class. Putting all of this together we get Figure 23.

## Static Structure



**Figure 23: UML – Structure: RealEstate sub hierarchy**

This diagram shows several new connections. First, a Player can navigate to its owned RealEstate objects. Each RealEstate object knows its owner, if any. Instances of the Utility class know about the PairOfDice; they need some kind of reference to the PairOfDice to calculate their taxes.

## Build 2: Code

### File: RealEstate.cs

```

1| namespace BoardGame {
2|     public abstract class RealEstate : BoardLocation {
3|         private int price;
4|         private Player owner;
5|         private bool hasLien;
6|
7|         public RealEstate(string name, int price) : base(name) {
8|             Assert.Argument(price > 0, "price", price, "should be > 0");
9|
10|             this.price = price;
11|         } // end RealEstate constructor
12|
13|         public abstract int Tax {
14|             get;
15|         } // end abstract Tax property
16|
17|         public int Price {
18|             get { return price; }
19|         } // end Price property
20|     }
  
```

```

21|     public int Value {
22|         get { return Price; }
23|     } // end Value property
24|
25|     public bool IsOwned {
26|         get { return Owner != null; }
27|     } // end IsOwned property
28|
29|     public Player Owner {
30|         get { return owner; }
31|     } // end Owner property
32|
33|     public bool IsOwnedBy(Player p) {
34|         return p == Owner;
35|     } // end IsOwnedBy method
36|
37|     public bool HasLien {
38|         get { return hasLien; }
39|         set { hasLien = value; }
40|     } // end HasLien method
41|
42|     public sealed override void LandOn(Player p) {
43|         if(!IsOwned) {
44|             OfferMyselfForSaleTo(p);
45|         } else if(!HasLien) {
46|             CalculateAndChargeTaxFor(p);
47|         } else {
48|             OnTurn(p.Name+ " avoided tax on " + Name, TurnSteps.LAND);
49|             // Nothing: No payments when there are Liens
50|         } // end if
51|     } // end LandOn
52|
53|     private void OfferMyselfForSaleTo(Player p) {
54|         p.Cash -= price;
55|         owner = p;
56|         p.AddRealEstate(this);
57|         OnTurn(p.Name+ " just bought " + Name, TurnSteps.LAND);
58|     } // end OfferMyselfForSaleTo method
59|
60|     private void CalculateAndChargeTaxFor(Player p) {
61|         if(!IsOwnedBy(p)) {
62|             int tax = Tax;
63|             p.Cash -= Tax;
64|             Owner.Cash += Tax;
65|             OnTurn(
66|                 p.Name + " just paid " + Owner.Name + " " + Tax + " tax ",
67|                 TurnSteps.LAND
68|             ); // end call to OnTurn
69|         } // end if
70|     } // end IsOwnedBy method
71| } // end RealEstate class
72| } // end BoardGame namespace

```

**File: City.cs**

```

1| namespace BoardGame {
2|     public class City : RealEstate {
3|         private int baseTax;
4|
5|         public City(string name, int price, int baseTax)
6|             : base(name, price) {
7|             Assert.Argument(baseTax > 0, " baseTax ", baseTax, "should be > 0");
8|
9|             this.baseTax = baseTax;
10|        } // end City constructor
11|
12|        public override int Tax {
13|            get { return baseTax; }

```



```
14|     } // end Tax property
15|   } // end City class
16| } // end BoardGame namespace
```

**File: Transportation.cs**

```
1| namespace BoardGame {
2|   public class Transportation : RealEstate {
3|     public Transportation(string name, int price) : base(name, price) {
4|     } // end Transportation constructor
5|
6|     public override int Tax {
7|       get { return 50; }
8|     } // end Tax property
9|   } // end Transportation class
10| } // end BoardGame namespace
```

**File: Utility.cs**

```
1| namespace BoardGame {
2|   public class Utility : RealEstate {
3|     private PairOfDice dice;
4|
5|     public Utility(string name, int price, PairOfDice dice)
6|       : base(name, price) {
7|       Assert.NotNull(dice, "dice");
8|
9|       this.dice = dice;
10|    } // end Utility constructor
11|
12|    public override int Tax {
13|      get { return 15 * dice.Value; }
14|    } // end Tax property
15|  } // end Utility class
16| } // end BoardGame namespace
```

**Updates: BoardLocationReader.cs**

```
1| // Add an attribute to hold on to the PairOfDice
2| private PairOfDice dice;
3|
4| // Update the constructor to take in the PairOfDice
5| public BoardLocationReader(string fileName, PairOfDice dice) {
6|   Assert.NotNull(fileName, "fileName");
7|   Assert.NotNull(dice, "dice");
8|
9|   suppliedFileName = fileName;
10|   this.dice = dice;
11|   Reset();
12| } // end BoardLocationReader constructor
13|
14| // Update the SetCurrentFrom method
15| private void SetCurrentFrom(string line) {
16|   String[] tokens = line.Split(SPLIT_CHARS);
17|   String type = tokens[0];
18|   String name = tokens[1];
19|   int price = 0;
20|
21|   switch(type) {
22|     case "start":
23|       current = new StartingLocation(name);
24|       break;
25|     case "city":
26|       price = Int32.Parse(tokens[2]);
27|       int baseTax = Int32.Parse(tokens[3]);
28|       current = new City(name, price, baseTax);
29|       break;
30|     case "random":
31|       current = new BoardLocation(name);
```

```
32|         break;
33|     case "surprise":
34|         int fine = Int32.Parse(tokens[2]);
35|         current = new SurpriseBill(name, fine);
36|         break;
37|     case "transportation":
38|         price = Int32.Parse(tokens[2]);
39|         current = new Transportation(name, price);
40|         break;
41|     case "bnb":
42|         current = new BoardLocation(name);
43|         break;
44|     case "utility":
45|         price = Int32.Parse(tokens[2]);
46|         current = new Utility(name, price, dice);
47|         break;
48|     case "spring":
49|         current = new SpringBoard(name);
50|         break;
51|     default:
52|         throw new
53|             ApplicationException("Unknown BoardLocation Type: " + type);
54|     } // end switch
55| } // end SetCurrentFrom method
```

### Updates: Player.cs

```
1| // We are now using collection classes, add a using statement for them
2| using System.Collections;
3|
4| // Add an attribute
5| private IList ownedRealEstate = new ArrayList();
6|
7| // Add a new method so that purchases RealEstate objects can be added
8| public void AddRealEstate(RealEstate r) {
9|     ownedRealEstate.Add(r);
10| } // end AddRealEstate method
11|
12| // Add a method so the Game can display a Player's RealEstate objects
13| public IList OwnedRealEstate {
14|     get { return ArrayList.ReadOnly(ownedRealEstate); }
15| } // end OwnedRealEstate method
16|
17| // Add a method so the Game can display how much a player is worth
18| public int TotalWorth {
19|     get {
20|         int sum = Cash;
21|         foreach(RealEstate r in ownedRealEstate) {
22|             sum += r.Value;
23|         } // end foreach loop
24|         return sum;
25|     } // end get section of TotalWorth property
26| } // end TotalWorth property
```

### Updates: Game.cs

```
1| // Update constructor to create BoardLocationReader with PairOfDice
2| BoardLocationReader r
3|     = new BoardLocationReader("BoardLocations.txt", myDice);
4|
5| // Update Play method to call PrintSummary method at the end of Play
6| public void Play(int rounds) {
7|     // Other code unchanged
8|
9|     PrintSummary();
10| } // end Play method
11|
12| // Add a method to display a summary at end of Play
13| private void PrintSummary() {
```

```

14|         foreach(Player p in players) {
15|             Console.WriteLine("{0} ends with {1}", p.Name, p.TotalWorth);
16|             foreach(RealEstate re in p.OwnedRealEstate) {
17|                 Console.WriteLine("\t{0}", re);
18|             } // end foreach loop over all RealEstate objects
19|         } // end foreach loop over all Player objects
20|     } // end PrintSummary

```

## Code Explained

### File: RealEstate.cs

#	Description
2	<p><b><i>public abstract class RealEstate : BoardLocation {</i></b></p> <p>Define a new class, RealEstate. It is a kind of BoardLocation object. This class will serve as an intermediate base class for other classes.</p> <p>This class is abstract, meaning it is not possible to create an instance of a RealEstate object directly. The only way to create a RealEstate object is to construct one of its subclasses.</p> <p>Attempting to write:</p> <pre>BoardLocation bl = new RealEstate("Name", 200);</pre> <p>Results in the following kind of compilation error:</p> <pre>MainApp.cs(7,22): error CS0144: Cannot create an instance of the abstract class or interface 'BoardGame.RealEstate'</pre>
3 – 5	<p>We have moved all attributes common across City, Transportation and Utility from City into RealEstate. Players can own instances of all three classes, can take liens out on the classes and each has a price. In build 1, City.cs describes a state attribute. This remains since it is not an attribute of either Utility or Transportation.</p>
7	Standard constructor.
13	<p><b><i>public abstract int Tax {</i></b></p> <p>This is our first abstract property. Abstract means that it must be defined in a subclass. If a subclass forgets to provide a non-abstract, or concrete, Tax property, C# will generate a compilation error. Abstract on a property also implies virtual.</p> <p>We know that the City, Transportation and Utility classes all have the concept of Tax. Each of them charges Tax to a Player when a non-owner lands on one of them. We do not have a reasonable default behavior because we need for each subclass to perform the calculation. By making this abstract, the class is enforcing an interface for subclasses.</p>
21	<p><b><i>public int Value {</i></b></p> <p>Define a new property called Value. This property returns the total value of a RealEstate object rather than its purchase price. Right now those values are the same. In later builds we will be changing this Property to change how we calculate the Value.</p>
42 – 51	<p><b><i>public sealed override void LandOn(Player p) {</i></b></p> <p>This is nearly a normal overridden method. The addition of the sealed keyword stops subclasses from writing their own version of LandOn. That is, now only does RealEstate force subclasses to write a Tax Property, it also forces them to not write a LandOn method.</p> <p>The remainder of this method is the same as when it was in City.cs. We have just moved it up to the intermediate base class because the algorithm is the same for all subclasses.</p>
53	<p>This method is nearly unchanged from City.cs. There is one difference. In Figure 23, we see a line between Player and RealEstate with an arrowhead on both sides. A RealEstate object knows its owner, if it has one. A Player knows all the locations it owns.</p>
56	The RealEstate objects informs the Player it is now the owner.
60	This method was moved from City up to RealEstate.
62	<p><b><i>int tax = Tax;</i></b></p> <p>This class only has an abstract Tax property. How can this code compile? C# realizes that this class is not complete because we made it abstract. C# will also force a subclass to provide a Tax</p>

#	Description
	property (or itself be an abstract class). When C# executes RealEstate.LandOn, the dynamic type of the receiver will be either City, Transportation or Utility. C# will make sure to use the correct Tax property.

**File: City.cs**

#	Description
2	<code>public class City : RealEstate {</code> City is no longer a subclass of BoardLocation. Instead it is a subclass of RealEstate. Much of its code has moved up to RealEstate.
5	As we have seen with constructors in other subclasses, this constructor calls its base class constructor. When we create an instance of city, the order of constructors is: <code>BoardLocation.BoardLocation(string name)</code> <code>RealEstate.RealEstate(string name, int price)</code> <code>City.City(string name, int price, int baseTax, string state)</code>
12	<code>public override int Tax {</code> City is providing its own Tax property. If it does not, C# will not compile the class. This version of tax simply returns the baseTax value. Eventually it will become more complex.

**File: Transportation.cs**

#	Description
2	Transportation is a subclass of RealEstate
3	It has no attributes other than the ones it inherits from its base classes.
6	It provides an implementation of the Tax property.

**File: Utility.cs**

#	Description
3	Utility holds a reference to a PairOfDice object, which it uses to calculate taxes. To better understand why Utility has this attribute, read “Background: Object Visibility” from page 118.
5	This constructor takes three parameters. The constructor passes two of parameters up to the base class and it stores one parameter in an attribute.
12	This Tax property gets the Value of the dice, multiplies that by 15 and returns the result. In all three cases we have enough of a hook to make Tax calculations as complex as we need to.

**Updates: BoardLocationReader.cs**

#	Description
2	BoardLocationReader now holds on to the PairOfDice object. It needs to keep this as an attribute because it does not use PairOfDice until it creates an instance of a Utility class. The first Utility class is several lines into the BoardLocations.txt file.
5	The constructor now requires the PairOfDice object.
19	We had to move the price local variable outside of the switch statement because we use it when constructing City, Transportation and Utility. We could have used additional {} to introduce nested scope but moving the variable out seems easier.
38	We create a Transportation object. We have to parse the price here. We cannot parse the price outside of the switch statement because there is only a price in the tokens array at index 2 for City, Transportation and Utility.
46	When we actually create the Utility class, we must pass in the PairOfDice object. The BoardLocationReader holds onto it in an attribute called dice.

**Updates: Player.cs**

#	Description
2	<p>The Player class knows all of the RealEstate objects it owns. We can see this from Figure 23 by the solid line between Player and RealEstate.</p> <p>To record this we could use an Array. However, how many RealEstate objects does a player own? This amount varies during game play, so rather than trying to use an array, we use some kind of collection object.</p> <p>Collection objects are in the System.Collections namespace.</p>
5	<pre>private IList ownedRealEstate = new ArrayList();</pre> <p>Define a private attribute that is a kind of IList object. IList is an interface. Interfaces are pure descriptions, they have no implementation. For now it is safe to think of IList as a super class of ArrayList. It is not but it serves a related purpose.</p> <p>An ArrayList is a class that holds on to objects. It is like an array, thus the name. You can add to the array list and it will grow if necessary. You can also remove things from the array list. It is a convenient way to record a one to many relationship.</p>
8	<pre>public void AddRealEstate(RealEstate r) {</pre> <p>This is the method that RealEstate objects use to inform the Player they are its owner.</p>
9	<pre>ownedRealEstate.Add(r);</pre> <p>Adding to our collection of RealEstate objects is easy, use the Add() message.</p>
13	<pre>public IList OwnedRealEstate {</pre> <p>Define a Property called OwnedRealEstate. This property returns an IList. It might be necessary for some object that uses Player to look at the Player's collection of RealEstate objects. Returning the whole collection can be quite dangerous. If a client of Player uses the property and then adds items to the collection, the collection will change without the Player knowing it. This may not seem like a big deal but we have seen projects essentially fail because of doing just this thing.</p> <p>It is not as bad as it might seem, however. As we will see in the next line, the Player returns a read-only version of its ArrayList. If someone tries change the collection's contents, the collection will throw an exception.</p> <p>In general, if one object maintains a collection of any kind, it should never allow direct access to the collection. Providing read-only access is acceptable. If the contents need manipulating, then add methods to the object to add or remove items from its collection. We have done this in line 12.</p>
14	<pre>get { return ArrayList.ReadOnly(ownedRealEstate); }</pre> <p>Use the static method ArrayList.ReadOnly to return a read-only wrapper around the collection. The wrapper object will throw an exception if someone tries to do anything that would add or remove items from the collection.</p>
18	<pre>public int TotalWorth {</pre> <p>We added this property so we could display the total value of each player at the end of the game. Game.PrintSummary uses this method.</p>
20	<pre>int sum = Cash;</pre> <p>Create a sum and set it equal to the cash the player has.</p>
21	<pre>foreach(RealEstate r in ownedRealEstate) {</pre> <p>Iterating over a collection looks no different than iterating over an array.</p>
22	<pre>sum += r.Value;</pre> <p>Get the value of each of the RealEstate objects owned by this player and add it to the running sum.</p>
24	<p>The TotalWorth of a player is equal to their Cash plus the Value of each RealEstate object they own. Note that we do not use the Price property. We are interested in how much the RealEstate object is worth, not what was its original cost.</p>

**Updates: Game.cs**

#	Description
2	Updated the creation of BoardLocationReader to pass in the PairOfDice object.
9	Added a call to a newly added PrintSummary() method to show some results after the game. We should add this to the TextUI, and we will. That is the purpose of one of the builds, so we are adding this in now to be able to see some of the effects of the changes.
13	A method to display the total value of a player. We have this here for now because we will need to make too many changes to our output to support printing a summary. Later we will update our system to have a proper user interface.
14	Iterate over each player. For each player we are going to print their total worth and a list of all of the RealEstate objects they own.
16	<i>foreach(RealEstate re in p.OwnedRealEstate) {</i> The OwnedRealEstate property provides a read-only version of the collection of RealEstate objects a Player owns. Iterate over the collection so we can print each out.
17	<i>Console.WriteLine("\t{0}", re);</i> Display each of the RealEstate objects.

**Build & Execute Instructions**

1. Create RealEstate.cs. It might be quicker to simply rename City.cs to RealEstate.cs and then update the copied file. If you do:
  - Replace all occurrences of “City” with “RealEstate”.
  - Remove the baseTax attribute.
  - Update the constructor to not take in baseTax and to remove the Assert as well as the assignment of baseTax.
  - Update the Tax property so that it is abstract and has no implementation.
2. Update City.cs. Make sure it inherits from RealEstate instead of BoardLocation. It is probably best to simply rename as suggested in step 1 and then create a whole new file.
3. Update Player.cs. Add the ownedRealEstate attribute, AddRealEstate method and the OwnedRealEstate and TotalWorth properties.
4. Compile these changes files and remove any errors before continuing.
5. Create Transportation.cs and Utility.cs. Make sure this compiles before continuing.
6. Update the constructor in BoardLocationReader.cs to take in a PairOfDice object as well as the switch statement in the SetCurrentFrom method.
7. Update Game.cs. When creating the BoardLocationReader, make sure to pass in the myDice attribute. Update the Play method to call the private PrintSummary method. Add the PrintSummary method.
8. Compile and execute:

```
c:\C#\BoardGame>csc *.cs
c:\C#\BoardGame>MainApp 2 100
Player#0 on StartingLocation(Starting Location) has 1500
    rolls 7
    will land on BoardLocation(Random Effect)
    and now has 1500
Player#1 on StartingLocation(Starting Location) has 1500
    rolls 3
    will land on City(Des Moines)
    Landing: Player#1 just bought Des Moines
    and now has 1440
Player#0 on BoardLocation(Random Effect) has 1500
    rolls 7
```

```
will land on City(Phoenix)
Landing: Player#0 just bought Phoenix
and now has 1340
Player#1 on City(Des Moines) has 1440
rolls 10
will land on City(Sedona)
Landing: Player#1 just bought Sedona
and now has 1300
```

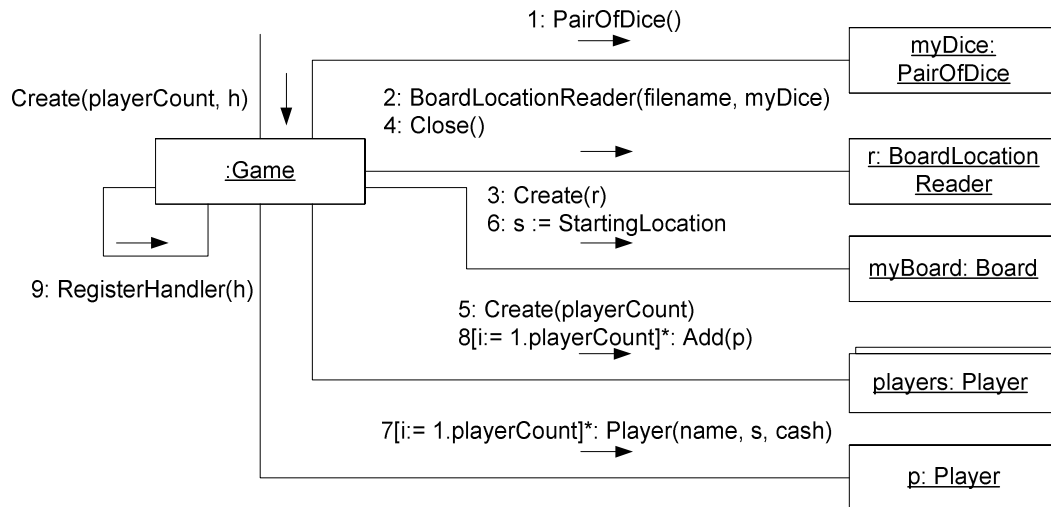
...SNIP...

```
Player#0 ends with 4949
City(Phoenix)
City(Chicago)
City(Alexandria)
City(Yakima)
Transportation(Air)
City(Scottsdale)
City(New Orleans)
City(San Francisco)
City(Houston)
Transportation(Taxi)
Transportation(Bus)
Transportation(Rail)
City(Montpellier)
City(Olympia)
City(New Port)
Player#1 ends with 4626
City(Des Moines)
City(Sedona)
City(Springfield)
City(Iowa City)
City(Dallas)
City(Rockford)
Utility(Green Electricity)
City(Bennington)
Utility(Natural Gas)
City(Baton Rouge)
City(Austin)
City(Redmond)
City(Sacramento)
```

## Object Collaboration

Most of the changes to our system are between Player and RealEstate. The overall flow of the game is unchanged from the previous build and iteration.

Construction of the Game changes only in one way, the Game passes its myDice object into the constructor of BoardLocationReader. Why? When a Utility object calculates its tax, it needs a reference to the PairOfDice object to be able to perform this calculation. For background on why we chose this design, read the background section on Object Visibility.

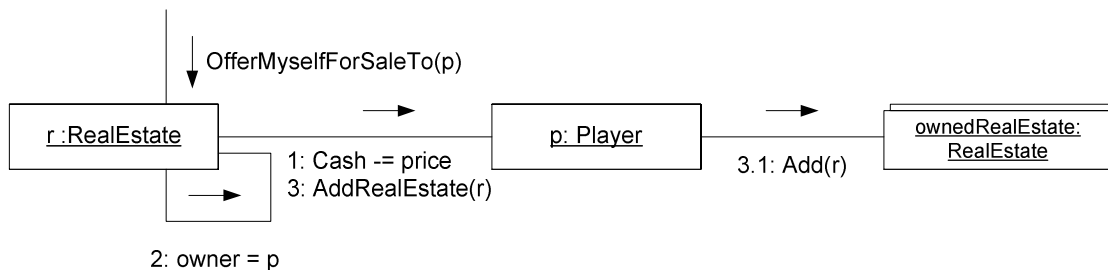
**Figure 24: UML – Collaboration: Game.Create**

????Where do I show the code for this??? I changed it so I only call StartingLocation once.

When a player lands on a RealEstate, one of three things happens:

- If the RealEstate object is not owned, it sells itself to the player.
- If the RealEstate object is owned, then it charges Tax if the current player is not its owner.
- If the RealEstate has a lien against it, it does nothing.

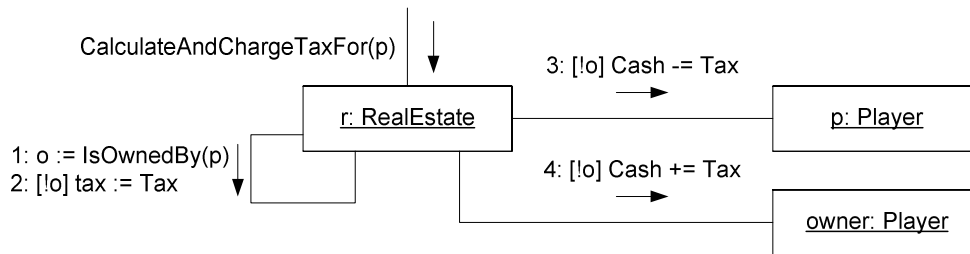
We can use somewhat complex notation and show all three of these paths of execution on the same diagram. Instead we will just look the first two individually. These following two diagrams start after the Player has sent the message `LandOn` to one of the subclasses of `RealEstate`.

**Figure 25: UML – Collaboration: RealEstate.OfferMyselfForSaleTo**

There are a few things to mention about this diagram. First, it is probably overkill. Second, it is technically incorrect to show the message `OfferMyselfForSaleTo` emanating from somewhere outside of the object and going into the object. If we review the code, the `LandOn` method sends this message to the current instance, which we normally represent using a loop like the one for step 2. Finally, remember that `RealEstate` is abstract. In practice we cannot have an instance of the `RealEstate` class, but this is a correct way to show the diagram.

For now, when a `Player` lands on a `RealEstate` object that is not owned, the `RealEstate` object charges the `Player` its cost regardless of whether the player can pay the price. The `RealEstate` object then sets its owner and tells the player to take on ownership of it. For now, the `Player` simply adds the object to its collection of `RealEstate` objects.

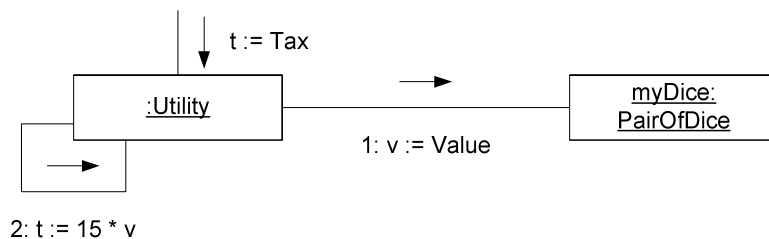




**Figure 26: UML – Collaboration: RealEstate.CalculateAndChargeTaxFor**

When a Player lands on RealEstate and it is owned, the RealEstate object sends itself the `CalculateAndChargeTaxFor` message to itself. The first thing this method does is check to see if the current player is the owner. If it is, nothing else happens. We can tell this because steps 2 – 4 have a guard on them. The guard, `[!o]` means: If `o` is false, then include this step. The variable `o` is set in step 1. It is true if the player passed in is the owner, false otherwise. Assuming the current player is not the owner, steps 2 – 4 calculate the tax, charge the current player and give the owner the money.

Notice that so far we have not examined any collaboration that is unique to any of the subclasses of `RealEstate`. The only part of this all of these interactions that changes is the calculation of Tax, step 2. The following diagram shows what happens for `Utility`. `City` and `Transportation` simply return a value so we do not include them here.



**Figure 27: UML – Collaboration: Utility.Tax**

The `Utility` object asks its `myDice` attribute for its value. It then multiplies that by 15 and returns the result. `City` and `Transportation` do not use the `PairOfDice` object to calculate their totals. Irrespective of what they do, `RealEstate.CalculateAndChargeTaxFor` can still do what it needs to do and it delegates the responsibility of Tax calculation to the subclasses.

As we introduce groupings of `City` objects, `Transportation` objects and `Utility` objects, we can update the Tax property to perform more complex calculation. Nothing anywhere else will change so we have managed to localize the effect of changing how we calculate taxes down to individual classes.

In build 3 we will add support for groupings of `RealEstate` objects and then update our Tax calculation algorithms.

### Build 3: Owning groups of RealEstate

When we start to consider grouping of `RealEstate` objects, the way we calculate Tax changes. The rules for the subclasses of `RealEstate` are:

Class	Description
City	If the same Player owns all of the cities in a State, taxes double. Once a player owns all of the Cities in the State they are additionally able to increase taxes by introducing higher state taxes.
Transportation	When the same player owns two Transportation objects, the Taxes double from 50 to 100. If the same player owns 3, Taxes are 200. All four and the taxes are 400.

Utility	If only one Utility is owned, taxes are 15x the current value of the PairOfDice. If both are owned, by the same or different players, taxes are 40x the current value of the PairOfDice.
---------	--

There are additional rules for Events, which we will look at in build 5. We will not look at increasing city taxes until build 6.

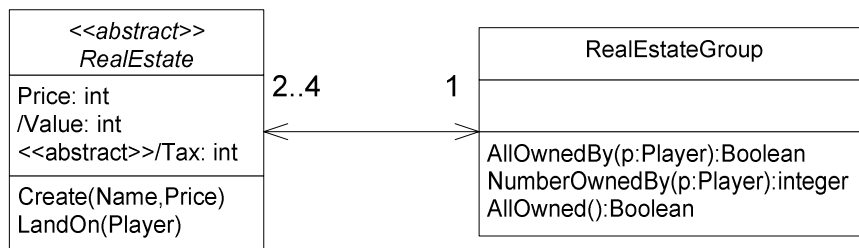
In all three cases, the taxes change based on groupings of RealEstate objects. If we look at the file containing all of the BoardLocation information (see “File: BoardLocations.txt” on page 51 ), we see that there are eight groupings of Cities, a grouping of Transportation objects and a grouping of Utilities. We need some kind of object that can maintain a group of RealEstate objects and answer some questions about them. Here are the questions we need to ask based on the class of grouping:

- City: Are all of the City objects owned and are they all owned by the same player?
- Transportation: What is the number of Transportation objects owned by the same player?
- Utility: Are all of the Utility objects owned?

We can solve this in several ways. Here are a few ideas:

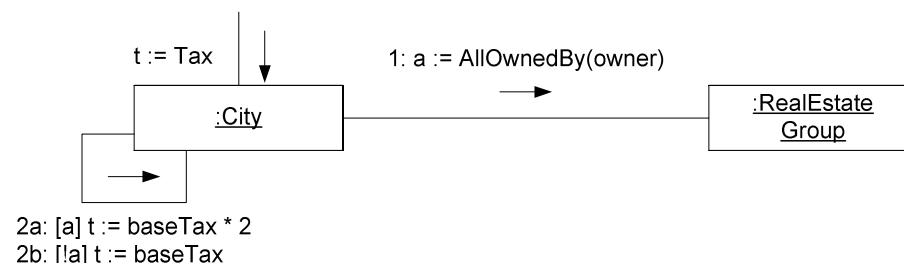
1. Create a single class that can answer all three of these questions. When we construct all of the boards, make sure we also create the necessary groupings of RealEstate objects.
2. Create a hierarchy of RealEstate groupings, one for each kind of RealEstate object, a so-called parallel hierarchy. Ask this class to modify the taxes based on whether the conditions necessary for increasing taxes have been met.
3. Create a single class that maintains the groupings and have it hold on to an algorithm that can perform the necessary calculations.
4. Create a single object that determines all of the rules for modifying the taxes and performs all of the necessary calculations.

### Option 1



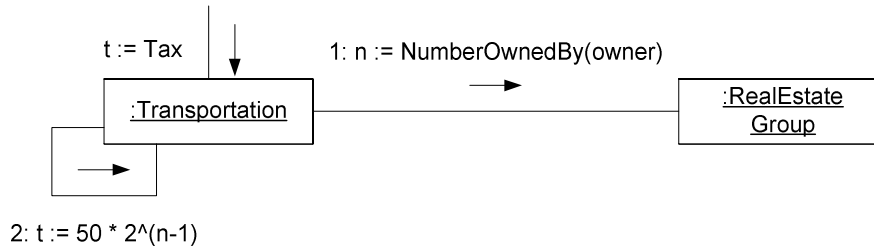
**Figure 28: UML – Structure: RealEstate & RealEstateGroup**

A RealEstate object can navigate to its RealEstateGroup. A RealEstateGroup knows all of the members of the group and can navigate to each one. Assuming this structure, calculating Tax for each of the three subclasses of RealEstate looks like the following:



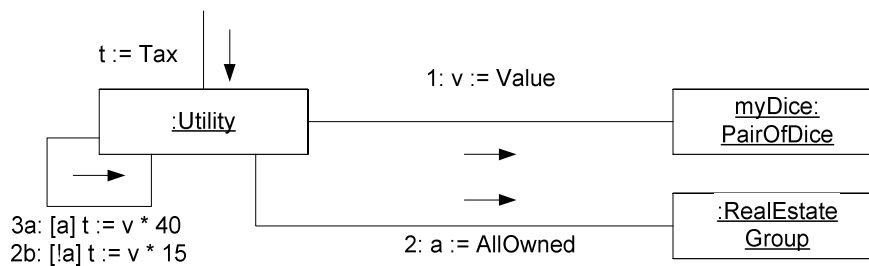
**Figure 29: UML– Collaboration: City.Tax**

If all of the group are owned and owned by the same player, step 1 returns true, otherwise it returns false. If step 1 returned true, then return  $\text{baseTax} * 2$ , otherwise return  $\text{baseTax}$ .



**Figure 30: UML – Collaboration: Transportation.Tax**

Ask the RealEstateGroup for the number of transportation objects owned by the same player. Multiply the basic Tax, 50, by 2 raised to the value of  $n - 1$ . This will give the values 50, 100, 200, 400 for value of  $n = 1, 2, 3, 4$ .

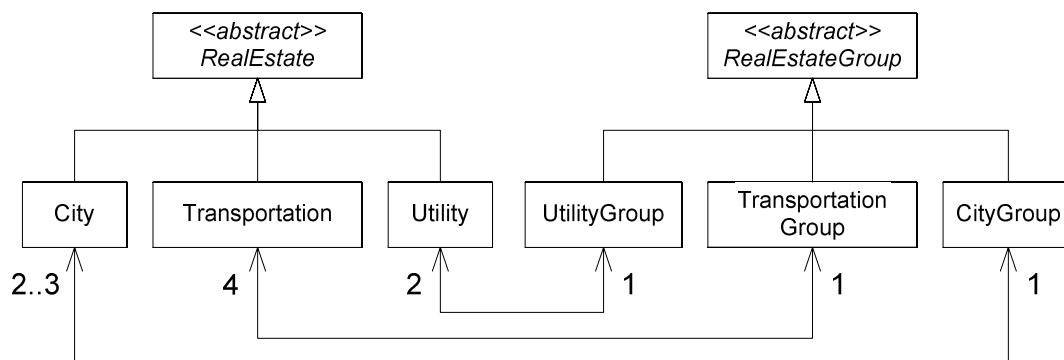


**Figure 31: UML – Collaboration: Utility.Tax**

Get the value of the PairOfDice object. Ask if all of the Utility objects are owned. If so, return  $40 * \text{the value from step 1}$ , otherwise return  $15 \text{ times the value from step 1}$ .

This option seems pretty straightforward assuming we can create the RealEstateGroup objects correctly. We can and to do so we will update BoardLocationReader. Ultimately this is the option we will use. We will look at the other options before moving on to code.

## Option 2



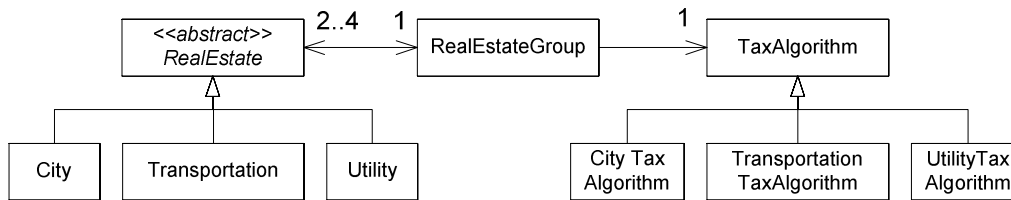
**Figure 32: UML – Structure: Parallel Hierarchy**

The static structure of this solution is already complex. Presumably if we were to add new kinds of RealEstate subclasses we would have to maintain a parallel hierarchy in RealEstateGroup. Without justifying this any further, we will nearly always avoid parallel hierarchies.

Rather than show the collaboration, we will simply describe it. In all three cases when asked to calculate Tax, each kind of RealEstate object will determine the base tax. It will then ask its

RealEstateGroup to update the Tax based on whatever rules apply. The code of the individual methods will look better than option one but the static structure is much worse.

### Option 3



**Figure 33: UML – Structure: Tax Algorithm**

This option actually might look as bad as option 2. Do not rule out this kind of solution but in this case the solution does not work. The common element between options 2 and 3 is that the way in which each kind of RealEstate subclass calculates its tax varies by the kind of RealEstate object. This might sound obvious but it is very important. This is an example where an obvious place to put the calculation is in each subclass of RealEstate. Because there is a direct solution to this problem, these other alternatives are not going to be as appealing.

### Option 4

We listed the fourth option because

- It is a possible solution,
- It is a common, albeit poor solution in general,
- And is a good solution if we happen to be using a rules engine.

We will not consider this option because the only way it works if we are not using a rules engine is by using conditional logic based on the type of the RealEstate object. This is in direct conflict with Polymorphism.

## Build 3: Code

### File: RealEstateGroup.cs

```

1| namespace BoardGame {
2|     using System;
3|     using System.Collections;
4|     public class RealEstateGroup {
5|         private string name;
6|         IList realEstatesInGroup = new ArrayList();
7|
8|         public RealEstateGroup(String name) {
9|             Assert.NotNull(name, "name");
10|
11|             this.name = name;
12|         } // end RealEstateGroup constructor
13|
14|         public void Add(RealEstate re) {
15|             realEstatesInGroup.Add(re);
16|         } // end Add method
17|
18|         public IEnumerator GetEnumerator() {
19|             return realEstatesInGroup.GetEnumerator();
20|         } // end GetEnumerator method
21|
22|         public string Name {
23|             get { return name; }
24|         } // end Name method
25|     }
  
```

```
26 |     public override string ToString() {
27 |         return GetType().Name + "(" + Name + ")";
28 |     } // end ToString method
29 |
30 |     public bool AllOwnedBy(Player p) {
31 |         foreach(RealEstate e in this) {
32 |             if(e.IsOwnedBy(p) == false) {
33 |                 return false;
34 |             } // end if
35 |         } // end foreach loop
36 |         return true;
37 |     } // end AllOwnedBy
38 |
39 |     public int NumberOwnedBy(Player p) {
40 |         int sum = 0;
41 |         foreach(RealEstate e in this) {
42 |             if(e.IsOwnedBy(p)) {
43 |                 ++sum;
44 |             } // end if
45 |         } // end foreach loop
46 |         return sum;
47 |     } // end NumberOwnedBy method
48 |
49 |     public bool AllOwned {
50 |         get {
51 |             foreach(RealEstate e in this) {
52 |                 if(e.IsOwned == false) {
53 |                     return false;
54 |                 } // end if
55 |             } // end foreach loop
56 |             return true;
57 |         } // end get section of AllOwned property
58 |     } // end AllOwned property
59 | } // end RealEstateGroup class
60 | } // end BoardGame namespace
```

#### Updates: RealEstate.cs

```
1 | // Add a new attribute
2 | private RealEstateGroup group;
3 |
4 | // Update constructor to take in RealEstateGroup
5 | public RealEstate(string name, int price, RealEstateGroup group)
6 | : base(name) {
7 |     Assert.Argument(price > 0, "price", price, "should be > 0");
8 |     Assert.NotNull(group, "group");
9 |
10 |     this.price = price;
11 |     this.group = group;
12 | } // end RealEstate constructor
13 |
14 | // Add Property
15 | public RealEstateGroup Group {
16 |     get { return group; }
17 | } // end Group property
```

#### Updates: City.cs

```
1 | // Update constructor to take in RealEstateGroup
2 | public City(string name, int price, int baseTax, RealEstateGroup group)
3 | : base(name, price, group) {
4 |     Assert.Argument(baseTax > 0, " baseTax ", baseTax, "should be > 0");
5 |
6 |     this.baseTax = baseTax;
7 | } // end City constructor
8 |
9 | // Update Tax property to use Group property
10 | public override int Tax {
11 |     get {
```

```
12|         if(Group.AllOwnedBy(Owner)) {
13|             return 2 * baseTax;
14|         } else {
15|             return baseTax;
16|         } // end if
17|     } // end get section of Tax property
18| } // end Tax property
```

#### Updates: Transportation.cs

```
1| // Update constructor to take in RealEstateGroup
2| public Transportation(string name, int price, RealEstateGroup group)
3| : base(name, price, group) {
4| } // end Transportation constructor
5|
6| // Update Tax property to use Group property
7| public override int Tax {
8|     get {
9|         return 25 * (1 << Group.NumberOwnedBy(Owner));
10|     } // end get section of Tax property
11| } // end Tax property
```

#### Updates: Utility.cs

```
1| // Update constructor to take in RealEstateGroup
2| public Utility(string name, int price, PairOfDice dice, RealEstateGroup g)
3| : base(name, price, g) {
4|     Assert.NotNull(dice, "dice");
5|
6|     this.dice = dice;
7| } // end Utility constructor
8|
9| // Update Tax property to use Group Property
10| public override int Tax {
11|     get {
12|         if(Group.AllOwned) {
13|             return 40 * dice.Value;
14|         } else {
15|             return 15 * dice.Value;
16|         } // end if
17|     } // end get section of Tax property
18| } // end Tax property
```

#### Updates: BoardLocationReader.cs

```
1| // We use a Hashtable to build group information
2| using System.Collections;
3|
4| // Add an attribute to store RealEstateGroup objects as they are created
5| private Hashtable groupsByName = new Hashtable();
6|
7| // Update the three cases dealing with groups
8| private void SetCurrentFrom(string line) {
9|     String[] tokens = line.Split(SPLIT_CHARS);
10|     String type = tokens[0];
11|     String name = tokens[1];
12|     int price = 0;
13|     RealEstateGroup group = null;
14|
15|     switch(type) {
16|         // unchanged cases removed.
17|         case "city":
18|             price = Int32.Parse(tokens[2]);
19|             int baseTax = Int32.Parse(tokens[3]);
20|             string state = tokens[4];
21|             group = GetOrCreate(state);
22|             current = new City(name, price, baseTax, group);
23|             group.Add((RealEstate)current);
24|             break;
```

```

25|         case "transportation":
26|             price = Int32.Parse(tokens[2]);
27|             group = GetOrCreate(type);
28|             current = new Transportation(name, price, group);
29|             group.Add((RealEstate)current);
30|             break;
31|         case "utility":
32|             price = Int32.Parse(tokens[2]);
33|             group = GetOrCreate(type);
34|             current = new Utility(name, price, dice, group);
35|             group.Add((RealEstate)current);
36|             break;
37|     } // end switch
38| } // end SetCurrentFrom method
39|
40| // Add support method for creating RealEstateGroup objects
41| private RealEstateGroup GetOrCreate(string name) {
42|     RealEstateGroup group = (RealEstateGroup)groupsByName[name];
43|     if(group == null) {
44|         group = new RealEstateGroup(name);
45|         groupsByName[name] = group;
46|     } // end if
47|     return group;
48| } // end GetOrCreate method

```

**File: RealEstateGroup.cs**

#	Description
3	<b><i>using System.Collections;</i></b> Figure 28 shows a one to many association between RealEstateGroup and RealEstate. The figure also shows the association is navigable in both directions, meaning every RealEstate object knows its RealEstateGroup and every RealEstateGroup knows all of its RealEstate objects. To represent this we will use a collection.
4	<b><i>public class RealEstateGroup {</i></b> This class does not belong to the BoardLocation hierarchy; it uses a part of the hierarchy. A common error would be to make this some how subclass either off of BoardLocation or RealEstate, but it only holds on to these objects, it cannot be used as one of these objects.
6	<b><i>ICollection realEstatesInGroup = new ArrayList();</i></b> This is the object we will use to collect all of the RealEstate objects in a group. We use the interface ICollection as the attribute type to allow us to easily change from ArrayList to some other kind of collection later. If for some reason we find there is a better kind of collection to use, we will change only this line (assuming the new kind of collection is also a kind of ICollection).
8	This class has a standard constructor. It does not call a base class constructor.
14	This method adds a new RealEstate object to this group.
15	<b><i>realEstatesInGroup.Add(re);</i></b> This actually adds a RealEstate object to the collection.
18 – 20	<b><i>public IEnumerator GetEnumerator() {</i></b> In “Updates: Player.cs” on page 82 we implemented a one to many association in Player. The player has a collection of RealEstate objects it owns. In Player there is a Property called OwnedRealEstate, which returns an ICollection. This allows us to look at each of the RealEstate objects owned by a player. To use this, we had the following code in the Game.PrintSummary(): <pre> foreach(RealEstate re in p.OwnedRealEstate) {     Console.WriteLine("\t{0}", re); } </pre> The Player has a collection of RealEstate objects. When we use foreach, we either provide an array or a collection object. We provided foreach a collection object by using the Player’s OwnedRealEstate property. A RealEstateGroup is a collection of RealEstate objects. It does not inherit from any collection

#	Description
	<p>class but we can still use it in a foreach expression directly. By providing a method called GetEnumerator, which returns an IEnumerator, we are able to use the RealEstateGroup object directly in a foreach statement. Assume we have a RealEstateGroup object called group. We can write the following:</p> <pre>foreach(RealEstate e in group) {     Console.WriteLine(e); }</pre> <p>What C# does is call the GetEnumerator method of group. If we wanted to use the IEnumerator directly, we could have instead written the following:</p> <pre>IEnumerator iter = group.GetEnumerator(); while(iter.MoveNext()) {     Console.WriteLine("\t{0}", iter.Current); }</pre> <p>Either works fine. There is an advantage with the foreach version. It converts what we get from the IEnumerator into a RealEstate for us. If you review the IEnumerator.Current property, its value is of type Object. For example, the following code will not compile:</p> <pre>IEnumerator iter = group.GetEnumerator(); While(iter.MoveNext()) {     RealEstate re = iter.Current; }</pre> <p>It will not compile because Current has a static type of “Object”. Its dynamic type is one of the subclasses of RealEstate: City, Transportation or Utility. We cannot automatically “down-cast” from Object to RealEstate. We instead have to use a cast to make this line compile:</p> <pre>RealEstate re = (RealEstate)iter.Current;</pre> <p>This tells C# that we think we know what is actually stored in Current. This happens to be a shortcoming of C#, its lack of support for generics or templates. Using foreach hides this from us by inserting the cast for us. It looks better although underneath the covers it is doing the same thing.</p>
22	This is a standard read-only property.
26	We override Object.ToString so we can print this object out and get human readable output.
30 – 37	City uses this method to determine if it should double its taxes. If this method returns true, City will double its taxes, otherwise it will return false. We iterate over each of the RealEstate objects in the group. If any of them are not owned by the Player objects passed it, return false. If we made it all the way through the collection then all of the RealEstate objects are owned by the same Player, return true.
39 – 47	Transportation uses this method to determine the total taxes. Start with a sum of 0. Iterate over each of the objects in the collection asking it if its owner is the Player passed in. If it is, increment the collection. Return the count at the end.
49 – 58	Utility uses this Property to determine if all of the members of the group are owned. If they are, Utility will change the amount by which it multiplies when calculating taxes. Iterate over each of members of the group. If we ever find one that is not owned, return false. If we make it all the way through, all of Utility objects are owned so return true.

**Updates: RealEstate.cs**

#	Description
2	<pre>private RealEstateGroup group;</pre> <p>Figure 28 shows a navigable relationship from RealEstate to RealEstate group with a multiplicity of 1 on the side of RealEstate. This means every RealEstate object knows its RealEstate group. We represent this association with an attribute.</p>
5	<pre>public RealEstate(string name, int price, RealEstateGroup group)</pre> <p>Change the constructor to take in a RealEstateGroup as the final parameter. Because we changed the constructor’s signature, we will have to change every constructor in every subclass</p>



#	Description
	of RealEstate. This is one of the costs of inheritance, so it is good to use inheritance only when it seems to help the solution overall.
15	Add a property to get the RealEstateGroup.

**Updates: City.cs**

#	Description
2	<code>public City(string name, int price, int baseTax, RealEstateGroup group)</code> Update the constructor to take in a RealEstateGroup, which it simply passed up to the base class on the next line.
10 – 18	Rewrite the Tax property. Now it uses its Group property. It asks the question, are all members of the group owned by the same player. If they are, multiply the baseTax by 2 and return that value. Otherwise return baseTax.

**Updates: Transportation.cs**

#	Description
3	Update the constructor to take an additional parameter and pass it up to the base class.
8 – 10	<code>get { return 25 * (1 &lt;&lt; Group.NumberOwnedBy(Owner)); }</code> This is a bit of a hack. The tax amounts are 50, 100, 200 and 400. Each time the number doubles. The result of asking for the NumberOwnedBy will be a value of 1, 2, 3 or 4. The << operator will take one and bit-shift it to the left. 1 << 1 results in 2, 1 << 2 results in 4, 1 << 3 gives 8 and 1 << 4 gives 16. 2 * 25 = 50, 4 * 25 = 100, 8 * 25 = 200 and 16 * 25 is 400. Multiplication, *, has a higher precedence than << so we have to use () to make the bit-shift happen faster.

**Updates: Utility.cs**

#	Description
2	Update the constructor to take in the new parameter.
10	Use RealEstateGroup.AllOwned to determine whether to multiple by 15 or 40.

**Updates: BoardLocationReader.cs**

#	Description
5	<code>private Hashtable groupsByName = new Hashtable();</code> BoardLocationReader must now handle groups of RealEstate objects. It will use a Hashtable to store the RealEstateGroup objects it creates. A Hashtable allows lookup based on another object. In this case we are going to look up the name of the group (a state name, transportation or utility) and get back the appropriate RealEstateGroup.
13	<code>RealEstateGroup group = null;</code> We are using a RealEstateGroup object for three of the case statements. We define the group reference here and use it in all of the cases that need it.
21	<code>group = GetOrCreate(name);</code> There are two situations with each RealEstateGroup object. Initially it does not exist. Once it does exist we want to make sure to use the same one we already had in the past. This private method will see if we already have a group by the provided name. If we do, use it. If we do not, then it will create it and return that group. BoardLocationReader will put every RealEstateGroup object it creates into the Hashtable defined on line 12.
22	<code>current = new City(name, price, baseTax, group);</code>

#	Description
	Create a City object and provide it with the group to which it belongs. Figure 28 shows that the association is bi-directional; RealEstate objects can navigate to their group, RealEstateGroup objects can navigate to all of their RealEstate objects. Passing the group into the City constructor sets relationship from RealEstate → RealEstateGroup.
23	<p><code>group.Add((RealEstate)current);</code></p> <p>Now we need to set up the association from RealEstateGroup → RealEstate. To do this we simply call RealEstateGroup.Add, passing in a RealEstate object. As soon as we do we hit a snag. We use a local variable called current. Current's static type is BoardLocation. This allows current to refer to an instance of BoardLocation or to any subclass of BoardLocation. This allows line 28 to compile.</p> <p>RealEstateGroup.Add expects a RealEstate object, not a BoardLocation. To get this line to compile we must tell C# to cast current to a RealEstate object. C# determines whether a given line will work or not at compile time and it does so looking only at the line and the static type information. Even though C# could look at the previous line to determine that current actually points to an instance of City, which is a kind of RealEstate object, it will never do this. A given line must compile on its own merits without looking at the program flow.</p> <p>In this case we know that current's dynamic type will be City and that casting it to RealEstate is safe. We do this and C# allows the line to compile. At runtime if for some reason current is not referring to a class that can be cast to RealEstate C# will throw an exception on this line.</p>
27, 33	The BoardLocation.txt file does not have group name information for either transportation or utility. Because of this we have simply use the type field, which will be either transportation or utility.
41	<p><code>private RealEstateGroup GetOrCreate(string name) {</code></p> <p>This private method will return the RealEstateGroup with the name provided. If none exists, this method will create it, remember it for next time and return the newly created object.</p>
42	<p><code>RealEstateGroup group = (RealEstateGroup)groupsByName[name];</code></p> <p>We will work in steps on this line. First we perform a lookup:</p> <p><code>groupsByName[name]</code></p> <p>The attribute groupsByName is our Hashtable. It has an indexed property, or an indexer as it is called in C#. When we access a property we generally use dot notation, e.g. aPlayer.Name. In this case we have a collection of objects. This collection knows objects by their name. To access this, the class exposes an indexer. To use an index we use []. This line will either return the object we previously stored under the provided name or it will return null if there is no object.</p> <p><code>(RealEstateGroup)groupsByName[name]</code></p> <p>Collections work with all kinds of objects. When we ask to get an object out of this collection, the collection returns back an Object. If you review the method signatures, they take in Object or return Object. This allows the methods to work with all kinds of objects. A better solution to this is to use templates but C# does not support templates. We therefore have to cast the result returned from the indexer to the type we think it is. If what we put into the collection does not match this type (or a subclass) then this part of the line will throw an exception.</p> <p>We finally assign the result to the local variable called group.</p>
43	<p><code>If(group == null) {</code></p> <p>Line 46 will set group to null if we have not yet created a group for the provided name. If the result is null we must construct a RealEstateGroup and store it in the collection.</p>
44	<p><code>group = new RealEstateGroup(name);</code></p> <p>Create the RealEstateGroup object and store it in the local variable group.</p>
45	<p><code>groupsByName[name] = group;</code></p> <p>Use the indexer to store the group object in the Hashtable. We used [] to both read from and write to the Hashtable. This only works because the Hashtable has defined a read-write indexer.</p>

#	Description
47	<i>return group;</i> Return either the group we found on line 46 or the one we created on line 48. The next time we look up the group with the provided name, we will find it using the indexer and return it.

## Build & Execute

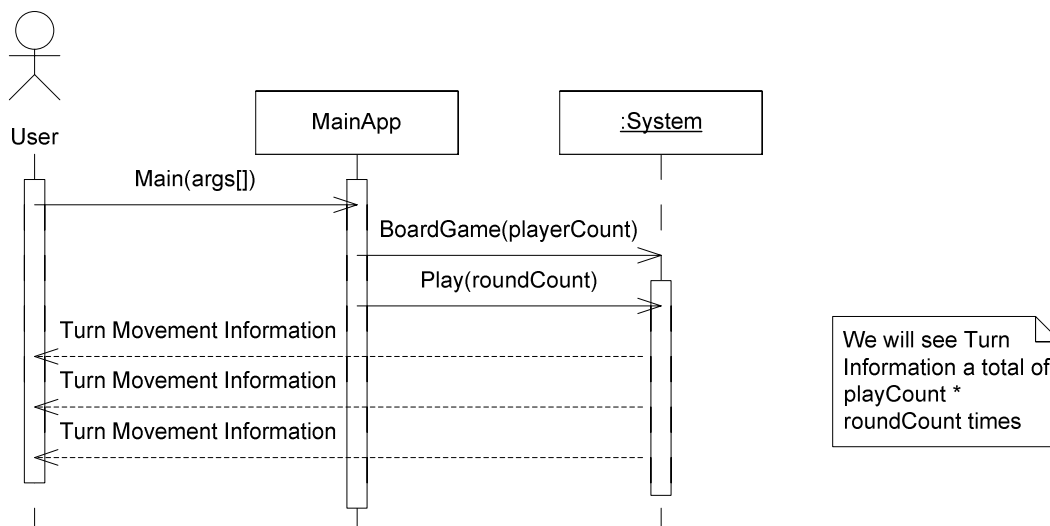
This is one of those situations where we have made a lot of changes that are interdependent. The initialization of the RealEstateGroup objects causes changes to construction. We can create RealEstateGroup and get it to compile first. Next, we will have to make the remainder of the changes because they are all interdependent.

1. Create RealEstateGroup.cs. Make sure it compiles before continuing.
2. Update RealEstate.cs to add the group attribute and Group property.
3. Update City.cs, Transportation.cs and Utility.cs. In all three cases you are doing the same two things:
  - Updating the constructor to take in a RealEstateGroup and passing it up to the base class.
  - Updating the Tax property to use the RealEstateGroup object to calculate taxes.
4. Update BoardLocationReader.cs. Change the case statements for City, Transportation and Utility. Add the GetOrCreate private method.
5. Compile and execute to make sure everything still runs.

## Build 4: A Simple Text UI

Up to this point we have managed to simulate the game and avoid user interaction. Builds 5 – 7 require players to make decisions. In addition, we have forced players to buy all of the RealEstate objects upon which they have landed, which does not follow the requirements.

We need to show some amount of user interaction. Before forging ahead into code, we will first take a look at some system sequence diagrams. Sequence diagrams are an alternative to collaboration diagrams; the two are interchangeable. Here is an example of what Main() looks like in our system:

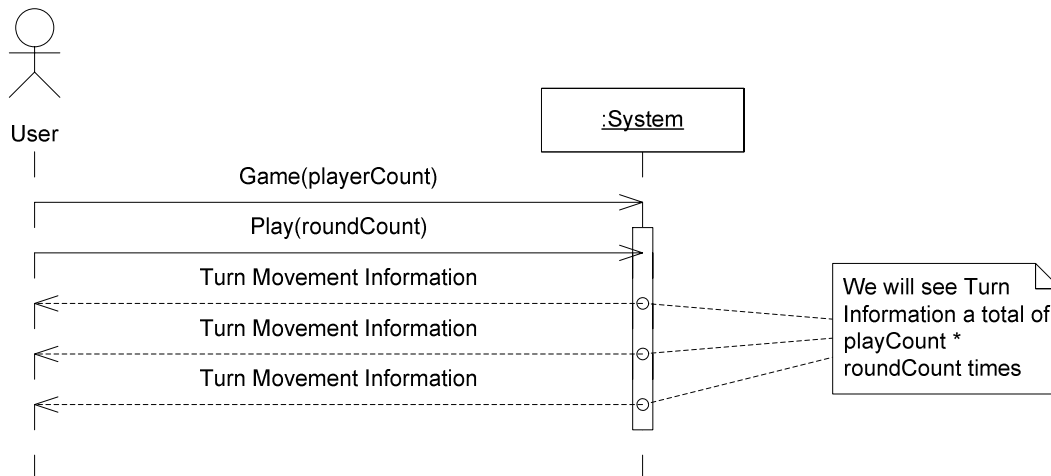


**Figure 34: UML – Sequence: The Game in Detail**

This shows that a User executes the program, `MainApp` in our case, which goes to the `MainApp` class (remember that `Main` is static, so the message is to the class and not an instance of the class). `Main`

reads command line parameters, which we have not shown, and then creates a Game object. It then tells the Game object to play a number of rounds. We see turn information for each player during each round of game play.

While it is permissible to use this level of detail on these diagrams, we will not do so in general. By convention we will only show the Actor interacting with the System and we will show the system as a black-box. The same diagram using this convention now looks like the following:



**Figure 35: UML – Sequence: Actor → System Level Detail**

Now that we have this, another point is that User above might be one or multiple users of the system. So if we have four players we will not show four Users on this diagram. It is certainly technically correct to do so we are just removing detail we do not see as necessary.

We are ready to review the requirements for this iteration and then develop a set of sequence diagrams to describe our system. Then we will redesign the system to handle user input and introduce the necessary design and code to allow user interaction using a simple textual interface.

## Requirements Reviewed

Right now when a player lands on or passes over locations something might happen. Before this iteration what happened was entirely in the control of the BoardLocation objects a Player passed over or landed upon. We introduced RealEstate but we simplified the solution to make the Player always purchase it. Now a player will be given the option to choose to purchase the RealEstate object or not.

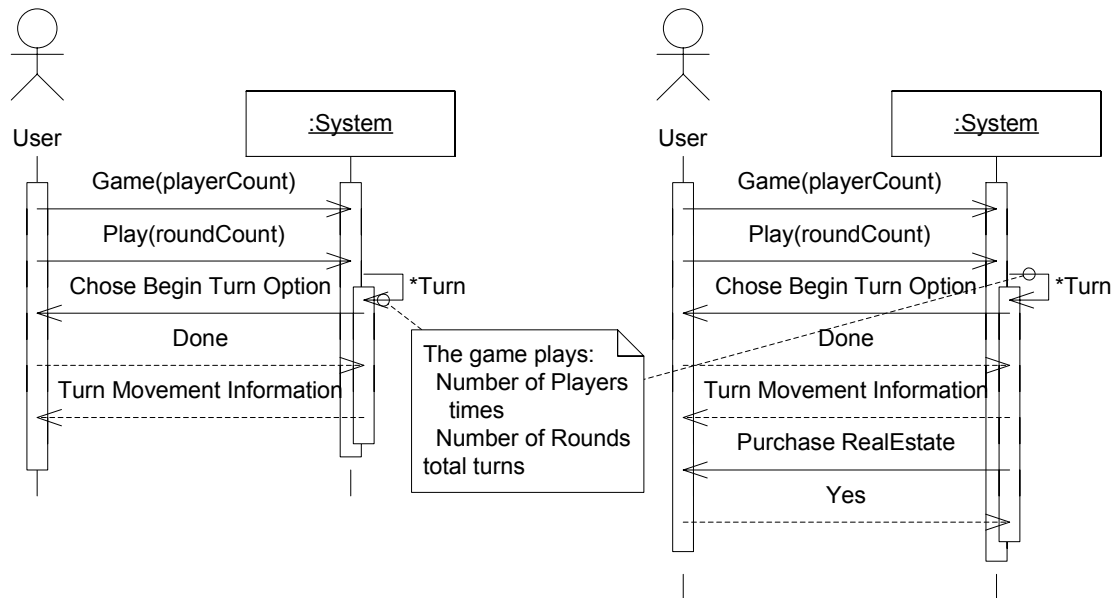
There are several more choices a player needs to make. At the beginning of a Player's turn, she might make any of the following choices before moving:

1. Take a lien out on a RealEstate object,
2. Pay back a lien on a RealEstate object,
3. Be forced to pay back a lien on a RealEstate object,
4. Plan a special event,
5. Increase city taxes,
6. Decreasing city taxes,
7. Stay on their current location (for Bed and Breakfast)

After making any of these choices any number of times, the Player then continues with her move. With our current system we cannot support any of these options. For this build we will simply add a menu with these options and then allow the player to choose to purchase a property.

## Actor to System Interaction

With this in mind, we have the following sequence diagrams:



**Figure 36: UML – Sequence: A Turn**

When a player takes a turn, the Game will present them with their begin turn options. Initially none of these will actually work so the only option will be for the player to finish their begin turn action. Next, the player will take what we had previously called taking a turn. A Player may have the option of purchasing a RealEstate object. Then the turn is over.

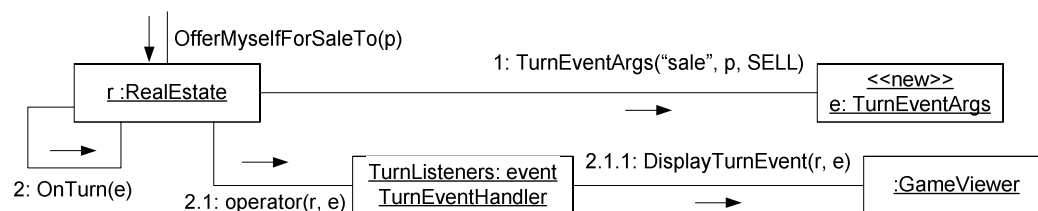
We need a design for each of messages from the user into the System:

## Actor to System Design

For now we are keeping changes to a minimum to support this new functionality. Right now we issue both of these system operations in the MainApp.Main.

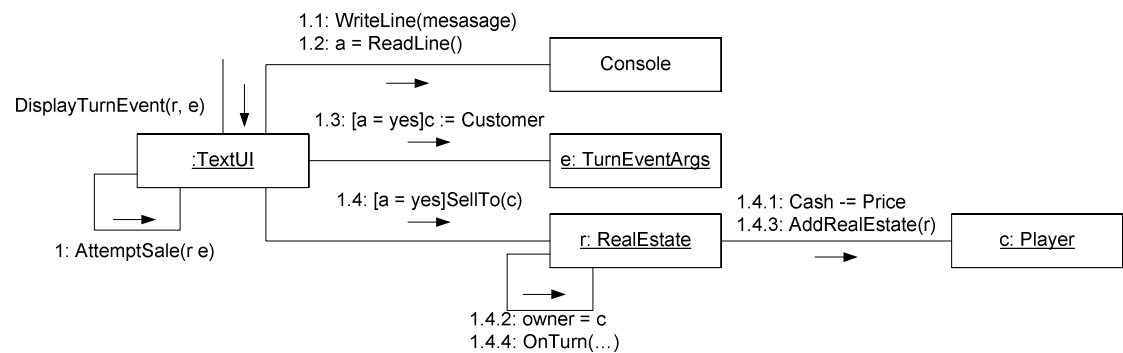
## Event Processing

During a turn a Player may be given the option of purchasing a RealEstate object. We use Events for output. Now the RealEstate object will generate an event and our TextUI will respond to that event by giving the actor a chance to purchase the RealEstate object. Starting in Build 1 we did this in a method called OfferMyselfForSaleTo. This method now resides in RealEstate and its most recent design is in Figure 25. With this update, we have the following redesign:



**Figure 37: UML – Collaboration: RealEstate.OfferMyselfForSaleTo(p: Player)**

The RealEstate object sends itself the message OfferMyselfForSaleTo. This method creates an event args object and then broadcasts a SELL event. This event makes its way to the TextUI object. The TextUI will ask if the player wants to purchase the location and if so, inform the RealEstate object:



**Figure 38: UML – Collaboration: TextUI.DisplayTurnEvent(Object, TurnEventArgs)**

TextUI responds to SELL events by calling an internal method. This internal method tells the user a RealEstate object is offering itself for sale and then asks the Player to decide on a purchase. Assuming the player chooses to purchase the RealEstate object, TextUI will get the Player from the TurnEventArgs using the Customer property and then tell the RealEstate object to sell itself to the player. The RealEstate object now does what it did before, the system is just asking for confirmation first.

**Updates: TurnEventArgs.cs**

```
1| // add SELL to enumeration
2| public enum TurnSteps { START, CONT, PASS, LAND, SELL, FINISH }
3|
4| // add a new attribute to TurnEventArgs
5| private Player customer;
6|
7| // add a new constructor, old constructor is still there.
8| public TurnEventArgs(string description, Player customer, TurnSteps action)
9| : this(description, action) {
10|     Assert.NotNull(customer, "customer");
11|
12|     this.customer = customer;
13| } // end TurnEventArgs
14|
15| // add Customer property to get new attribute
16| public Player Customer {
17|     get { return customer; }
18| } // end Customer property
```

#	Description
2	We have added “SELL” as one of the steps within a turn. The TextUI class will use an event with this action to ask if the user wants to buy the RealEstate upon which she just landed.
5	We now need to know the current Player when a SELL event happens. This attribute is not always required so it might be null. It will be the responsibility of the event source to properly fill in the TurnEventArgs object. We could have created different kinds of TurnEventArgs, we instead chose to have one class that handles all events, with the caveat that we know sometimes some of the attributes might be null.  Note that we could have gone one step further and added an owner attribute. Had we done that, then it would have been possible to simplify the description string we create everywhere. This was not necessary to get the build running so we will consider that in a refactoring later.
8	We added a new constructor. This constructor sets the customer attribute.
16	We added a Property to retrieve the customer attribute. Note that we called this attribute

#	Description
	customer to indicate its intended use.

**Updates: RealEstate.cs**

```

1 | // Rewrite OfferMyselfForSaleTo
2 | private void OfferMyselfForSaleTo(Player p){
3 |     TurnEventArgs args = new TurnEventArgs("sale", p, TurnSteps.SELL);
4 |     OnTurn(args);
5 | } // end OfferMyselfForSaleTo method
6 |
7 | // Add SellTo
8 | public void SellTo(Player p) {
9 |     p.Cash -= price;
10 |    owner = p;
11 |    p.AddRealEstate(this);
12 |    OnTurn(p.Name + " just bought " + Name, TurnSteps.LAND);
13 | } // end of SellTo method

```

#	Description
2	In previous builds and iterations, OfferMyselfForSaleTo assumed the player wanted to purchase the RealEstate object. Now it simply generates an event. If there happens to be an event receiver listening, it can use this event to determine whether the user wants to purchase the RealEstate object or not. We have taken the original functionality and placed it in SellTo.
3	Create a TurnEventArgs using the newly added constructor. When we switched to using events for output, we added a support method called OnTurn, which took a string and an action. We did not do that here simply because there is only one place where we use the new constructor so adding the support method would not have improved our solution.
4	Trigger the event. Use the standard form of OnTurn, which takes the TurnEventArgs.
8	SellTo is the original functionality from OfferMyselfForSaleTo. If the event receiver determines that a user wishes to purchase the RealEstate object, it will call this method to finish the transaction.

**Updates: TextUI.cs**

```

1 | // Add new case in DisplayTurnEvent
2 | case TurnSteps.SELL:
3 |     AttemptSale(sender, e);
4 |     break;
5 |
6 | // Add AttemptSale
7 | private void AttemptSale(object sender, TurnEventArgs e) {
8 |     RealEstate re = (RealEstate)sender;
9 |     Console.WriteLine("{0} is for sale, purchase it? ", re.Name);
10 |    string answer = Console.ReadLine().Trim().ToLower();
11 |    if(answer.StartsWith("y")) {
12 |        re.SellTo(e.Customer);
13 |    } // end if
14 | } // end AttemptSale method

```

#	Description
2	Add a new case to the switch statement. This case is for the SELL event action. Simply call the AttemptSale message.
8	When we generate an event, we pass two objects. The first is the event source, which we have called sender. The second is the TurnEventArgs. Given that any object in our system can generate a turn event, we use the base class object as the type of the sender. Since all classes either directly or indirectly inherits from object, a reference to an object can refer to anything. To put this another way, the static type object can refer to any dynamic type. Since we have an object but we know that RealEstate objects trigger SELL events, we can



#	Description
	“safely” downcast from object to RealEstate. By down casting, we can now use methods defined in the RealEstate class.
9	Output a message asking the user if they wish to purchase the RealEstate object.
10	<pre><i>string answer = Console.ReadLine().Trim().ToLower();</i></pre> <p>This line performs many steps. First it acquires a line from the input stream. It takes whatever the user types up to when they press return. The result is a string. Next, we Trim() the string, which removes white space from either side (spaces, tabs, new lines). We then use ToLower() to convert whatever remains into all lower-case letters. Finally, we store the result of all of this into the variable answer.</p> <p>In doing all of this we create three strings. String objects are immutable. When we use methods like ToLower or Trim, the original string is unchanged. We get back a newly created string instead.</p>
11	Check to see if the first letter of the input string starts with “y”.
12	It does, call the SellTo method of the RealEstate object, which then deducts money and sets up the relationships between Player and RealEstate.

## Build & Execute

1. Apply the changes to all three classes. Compile:

```
csc *.cs
```

2. Execute MainApp using enough rounds to make sure someone lands on a RealEstate:

```
MainApp 2 20
Player#0 on StartingLocation(Starting Location) has 1500
    rolls 4
    will land on SurpriseBill(Leak In Roof)
    Player#0 pays 100
    and now has 1400
Player#1 on StartingLocation(Starting Location) has 1500
    rolls 11
    will land on City(Scottsdale)
Scottsdale is for sale, purchase it? y
    Landing: Player#1 just bought Scottsdale
    and now has 1360
Player#0 on SurpriseBill(Leak In Roof) has 1400
    rolls 5
    will land on City(Austin)
Austin is for sale, purchase it? y
    Landing: Player#0 just bought Austin
    and now has 1280
Player#1 on City(Scottsdale) has 1360
    rolls 10
    Passing: Player#1 incremented pass count
    will land on City(Rockford)
Rockford is for sale, purchase it? y
    Landing: Player#1 just bought Rockford
    and now has 1140
```

## Build 5: Taking out a lien

A player may take out a lien at the beginning of their turn or in response to having to pay some amount of money. This introduces several changes:

- At the beginning of each turn, Players should be given the option to take out a lien any RealEstate they own that does not already have a lien.
- At the beginning of each turn, Players should be given the option to pay back a lien.
- At the beginning of a turn, a Player may be forced to pay back a lien. This happens if the lien amount is > 2x the original lien amount. If the player does not select the pay back lien option, the



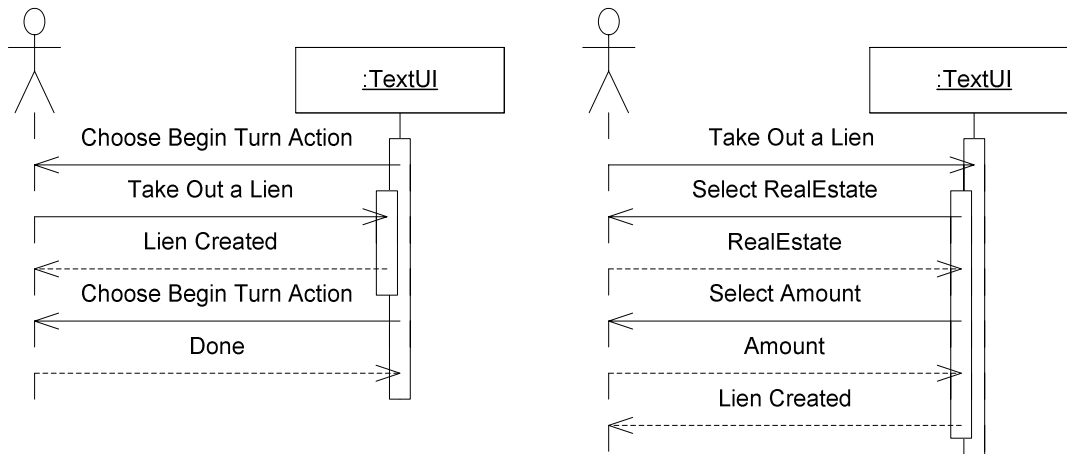
system will determine if the player has enough cash on hand to pay the lien. If so, the system will simply take the cash from the player to pay back the lien. If the player does not have enough cash on hand, the system will present two options:

- Pay back the lien by taking out other liens or selling off tax increases.
- Do not pay back the lien; lose what cash the player has on hand and the RealEstate object becomes Unowned.
- At any time during the game when Players are forced to pay some amount of money and do not have enough cash on hand, they should be given the option to take out a lien. One of two things happen:
  - The player pays the fine by taking out a lien.
  - The player is unable take out enough liens to pay the fine. The money still goes to the player owned taxes, all of the player's RealEstate becomes Unowned and the player is out of the game.
- At the beginning of each turn, all lien amounts increase by 8%, rounded down to the nearest integer value.

## Sequence Diagrams

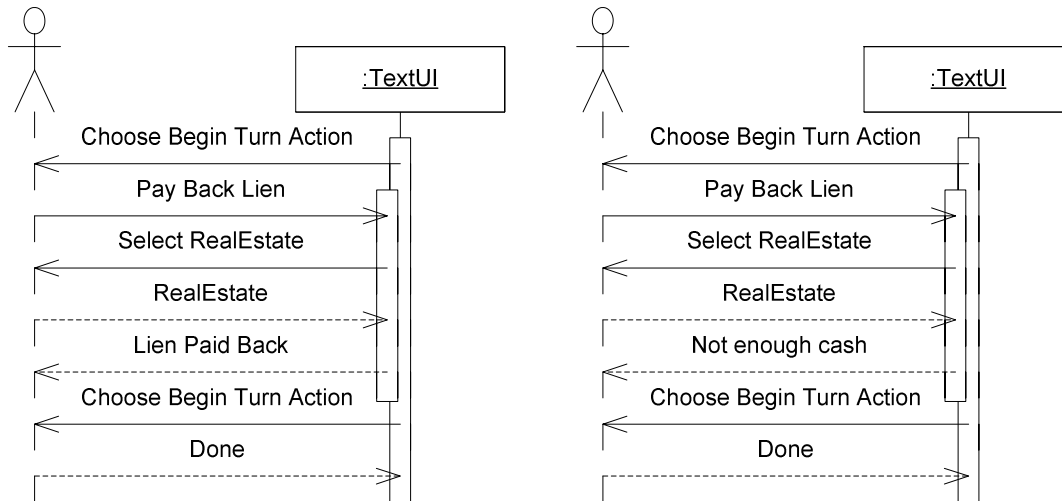
We will represent each of these scenarios where the system requires user interaction using a scenario diagram. We will show an actor interacting with the system, which we will call “:TextUI”. We will not show any of the internal details of what happens within the system, so these diagrams are system sequence diagrams.

First, we show taking out a lien. We break this into two sequence diagrams since taking out a lien is something which can happen both at the beginning of a turn and later. By showing taking out a lien as a separate diagram we can refer to it in other diagrams.

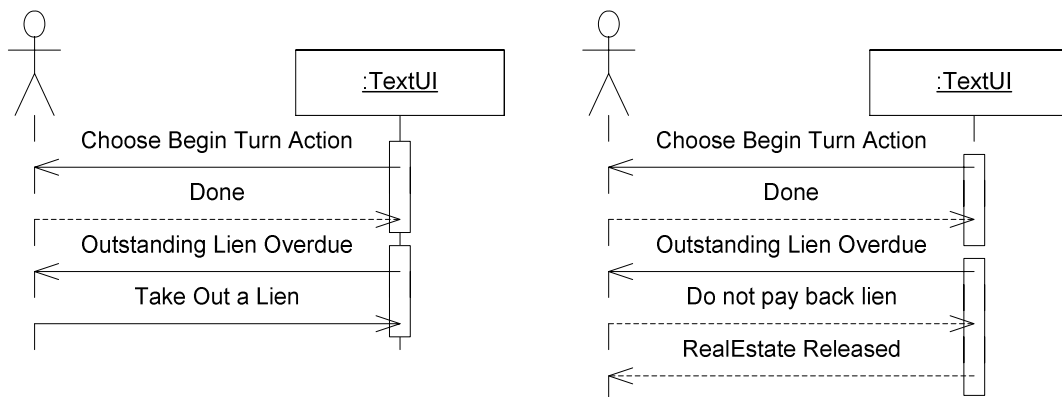


**Figure 39: UML – Sequence: Taking out a lien**

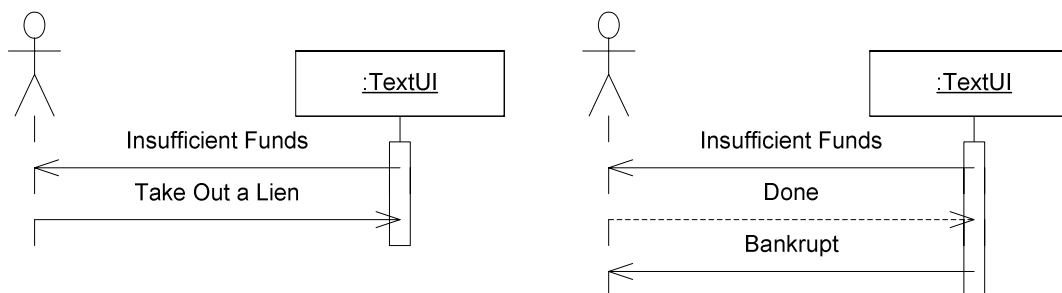
Paying back a lien may only be done at the beginning of a turn so we do not separate it into two diagrams. However we do have two diagrams because paying back a lien could fail if the player does not have enough cash on hand.

**Figure 40: UML – Sequence: Paying back a lien**

A player may be forced to pay back a Lien. If the player has enough cash, the system will automatically take it from them. If they do not, the system will force the player to either take out a lien to pay back what they owe or to choose to release the RealEstate object:

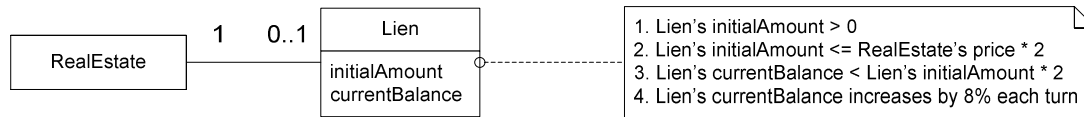
**Figure 41: UML – Sequence: Outstanding lien overdue**

Finally, a player may be charged taxes or some kind of fine and not have sufficient funds. When this happens, they will have the choice to sell back RealEstate. If they cannot raise sufficient funds, they are bankrupt and out of the game:

**Figure 42: UML – Sequence: Insufficient funds**

## Domain Model

While there are a lot of requirements for this build, there are not too many new concepts. The only one is a Lien. The following domain model shows how this new concept fits into what we already know about the system:



**Figure 43: UML – Static Structure: Partial Domain Model**

In this diagram we see that a RealEstate object may be associated with 0 or 1 Liens. A Lien is for exactly one RealEstate object. There are other rules about the Lien we have captured in the note box connected to the Lien concept. While we have used English to capture these rules, there is a more formal way called object constraint language (OCL). We have not used OCL to avoid introducing too much syntax.

## System Operations/Events

Using the sequence diagrams as our guide, we have several system operations and events for which we will create a detailed design. Before we attempt a design we should describe the system operations in more detail.

### Choosing begin turn actions

Figure 39, Figure 40, and Figure 41 show the system asking the user to chose a begin turn action. The user makes a choice, such as taking out a Lien, answers some further questions, and then gets back to choosing another begin turn action. This continues until the user indicates she is finished, whereupon the turn continues with the Player rolling the PairOfDice and moving.

#### Take out a lien

Figure 39 shows the steps for taking out a Lien. For this operation to be possible, what must be true before the operation?

- It must be the Player's turn.
- The Player has not yet moved or is being forced to pay a fine.
- The Player must own at least one RealEstate object that does not have a Lien.

Assuming all of these conditions are met, what will be the result of taking out a Lien?

- A new Lien object is created.
- The initialBalance and currentBalance of the lien will be equal and within the range of 0 to 2 \* price of the RealEstate object.
- The association between the Lien and the RealEstate object now exists.
- The Player will have one less RealEstate object that does not have a Lien on it.

The first list contains preconditions. The second list contains post-conditions. Strictly speaking, just before executing the operation, if all of the preconditions are met, then the system should guarantee all post-conditions are true after the operation. If all of the preconditions are not met, nothing can be said about the post-conditions.

#### Pay back a lien

Figure 40 shows the steps for taking out a Lien. For this operation to be possible, what must be the case before the operation?

- It must be the Player's turn.
- The Player has not yet moved.
- The Player must own at least one RealEstate object that has a Lien on it.
- The Player must have enough cash to pay back the current balance.

Assuming all of these conditions are met, what will the result of paying back the Lien?

- The Player's cash balance is reduced by the current balance of the Lien.
- The Lien object is destroyed.

As with the previous system operation, the first list contains conditions that must be true for the operation to be a success, so-called preconditions. The second list contains the results of the operation after it has happened, or post-conditions. Using both of these as checklists, we will show a design that handles both the preconditions and post-conditions.

### Outstanding lien overdue

Figure 41 shows what happens when a Player attempts to begin moving with an outstanding Lien. What must be the case for this to happen?

- The Player has indicated they are ready to move but has not yet moved.
- The Player must own at least one RealEstate object that has an overdue Lien on it.
  - Overdue means the balance is  $\geq 2$  times the initial amount of the balance.
- The Player's available cash must be less than the amount due.

After this operation one of two things has happened either the Player took out another Lien on another RealEstate object to pay off the overdue Lien or they chose to release the RealEstate object. In later builds the Player will be able to sell off City tax increases to pay back Liens, but we have not got that far yet. We will show post-conditions for both of these:

- Player pays back Lien
  - The Player has at least one more Lien and the Player's cash balance is increased by the Lien amount.
  - The Player's cash balance is reduced by the amount of the Lien's current balance
  - The Lien is destroyed.
- Player releases RealEstate
  - The Player's cash balance = 0.
  - The RealEstate object no is no longer owned.
  - The Lien on the RealEstate is destroyed.

### Insufficient funds

Figure 42 shows the circumstance when a Player is forced to pay a fine but has insufficient funds. What are the pre-conditions for this operation?

- It is a Player's turn and they have already moved and have landed.
- They were charged some kind of fine whose amount was more than the Player's cash balance.
- The Player is able to pay back the fine by taking out Liens.

Two things might happen here, either the Player can pay back the fine by taking out a Lien, or they are bankrupt and out of the game. We have added the last of the pre-conditions so we can have one set of post conditions. After this operation we have the following post-conditions:

- The Player has taken out at least one new Lien.
- The total of all Liens the Player took out plus the Player's cash amount  $\geq$  fine.

If the Player was unable to pay back the fine, she is out of the game and all of her RealEstate objects revert to having no owner.

### Begin turn event

Referring back to Figure 43, we need to confirm that we have captured all of the notes in the note box. Reviewing the list we see the following:

- We covered items one and two in the post-conditions of “Choosing begin turn actions”.
- We covered the third item indirectly with “Pay back a lien”.
- We covered the third item directly with the “Outstanding lien overdue” system operation.
- We have not addressed item four.

We missed the fourth item because it is something that happens behind the scenes. Conceptually, we need something to happen at the beginning of the turn, but before the Player has the option of doing anything. This “Begin Turn Event” has no preconditions for this operation, it simply happens. There is one post-condition:

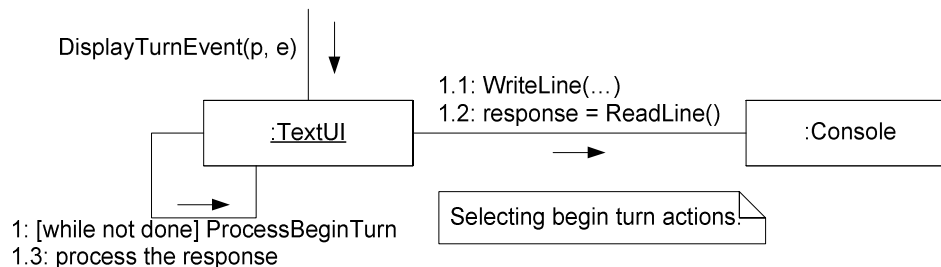
- All Liens on all of the RealEstate objects owned by the Player increase by 8% rounded down.

## Detailed Design Using Collaboration Diagrams

We have described the system operations and events sufficiently to allow us to move on to detailed design. This section shows the detailed design for each of our system operations and events. We will design each of the operations in the order we describe them above.

### Choosing begin turn actions

In our current design, the Game object tells the Player object to TakeATurn(). The Player object triggers an event indicating the beginning of its turn. The TextUI should now prompt the user for any begin turn actions.

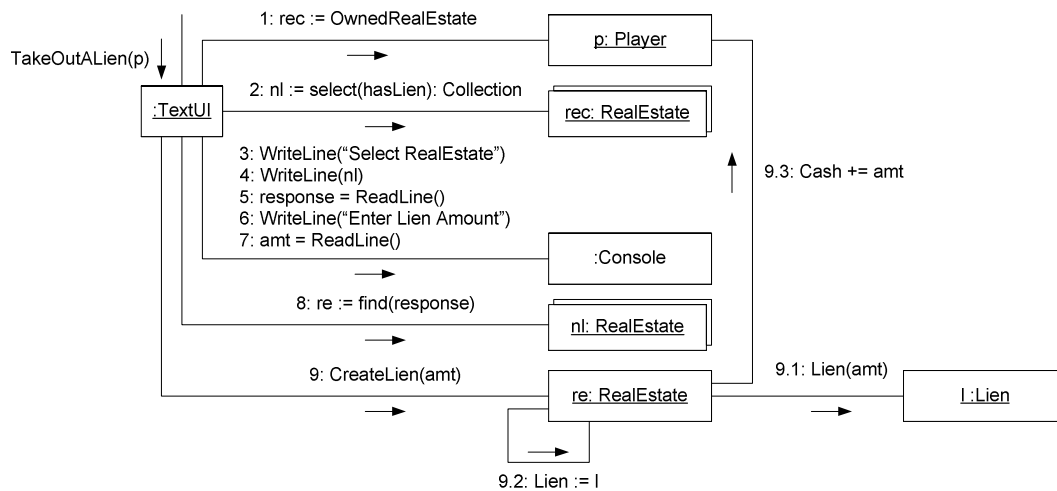


**Figure 44: UML – Collaboration: Choose begin turn actions**

When the TextUI is told to `DisplayTurnEvent()` then happens to be the START of a turn, it will repeatedly ask the user what she want to do. The user will make a selection; the TextUI will process that selection. This continues until the user indicates she is done.

### Take out a lien

One option a user might select is taking out a Lien. This can also happen in response to insufficient funds. When this happens, we see the following:



**Figure 45: UML – Collaboration: TakeOutALien(Player)**

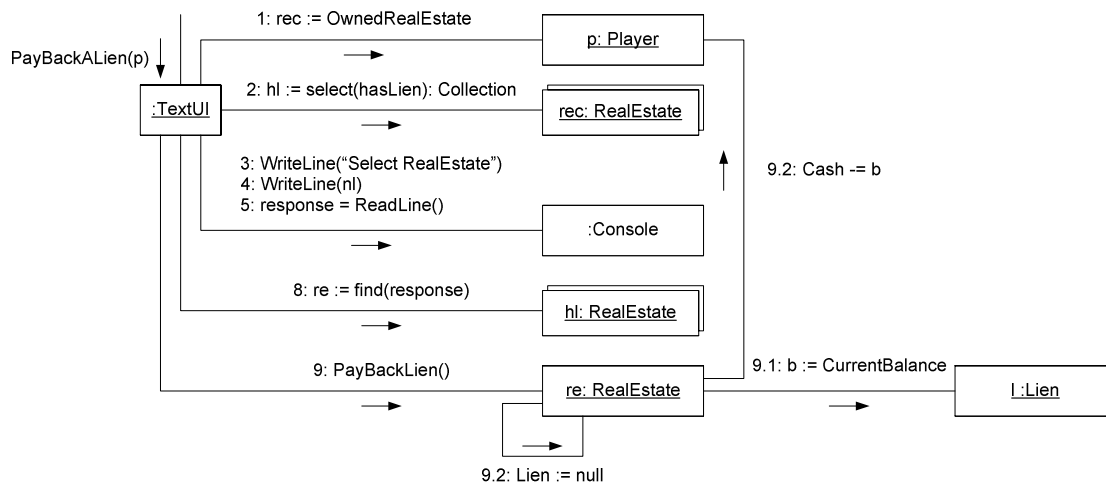
When a user requests to take out a Lien, the TextUI will send itself the message TakeOutALien. The TextUI will get all RealEstate objects owned by the current Player. The TextUI then finds only those RealEstate objects that do not already have a Lien against them. The TextUI next prompts the user, asking her to select one of the RealEstate objects and then the TextUI a Lien amount. With all of the necessary information, the TextUI finds the particular RealEstate object the user selected and sends it the message createLien. The RealEstate object creates a Lien object, records it and then updates the Player's balance to reflect the Lien.

This diagram does not show all of the details for the interaction. For example, we do not show that if we try to create a Lien on the RealEstate for an invalid amount, the RealEstate object will throw an exception. We already have a general policy regarding parameter validation so we do not show that in this diagram.

We also do not show checking for an existing Lien on the RealEstate object. Remember that there can be at most 1 Lien object associated with a RealEstate object. You might ask why check? The TextUI only shows those RealEstate objects that do not already have a Lien. While this TextUI does that, not checking actually puts rules for the game within the User Interface. This means that if we later develop another user interface, which we will be doing, the alternative user interface will also need to validate that game rule. The rules for the game should be in the game, not the user interface.

### Pay back a lien

A user may choose to pay back a lien. This looks very similar to our previous interaction:

**Figure 46: UML – Collaboration: PayBackALien(Player)**

This collaboration starts much the same as the previous one. We get the Player’s RealEstate objects and this time we select only those that have a Lien. We then ask the user to select a RealEstate object. The user makes a selection. The TextUI looks up RealEstate object based on the user’s response and sends the resulting object the message PayBackLien. The RealEstate object determines the Lien balance, charges the Player the balance and then removes the Lien against it.

As with the previous example, we do not show all of the exception processing. For example a RealEstate object will throw an exception if we ask it to PayBackLien and it does not have one. The Player will also throw an exception if we ask them to pay some amount of cash and the do not have enough cash on hand.

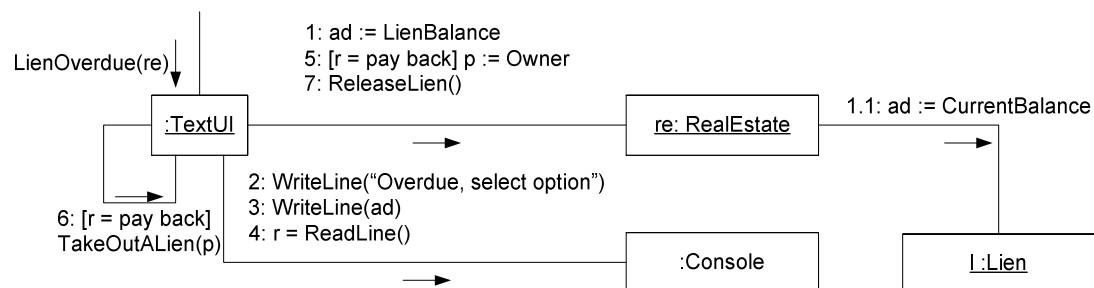
### Outstanding lien overdue

We have called this an operation but it really is an event. This event occurs when a Player attempts to begin moving but has not paid back overdue Liens. We are missing an event; the Player begins a turn, starts moving, passes over several locations, lands on a location and finally finishes her turn.

This event occurs after beginning a turn and just before a player starts moving. We will need to update our event triggering. This is something we will at to a list of “Remaining to do’s” and cover that after we have addressed all the system operations.

Assuming our events happen as required, the TextUI will receive an event that a Lien is past due and process it accordingly:

<<working with RE, but should be getting LIEN>> Lien will need to know its realestate.

**Figure 47: UML – Collaboration: LienOverdue(RealEstate)**

The TextUI begins by getting the current Lien balance. TextUI asks the RealEstate object, which asks its Line object for the current balance. The TextUI then tells the user about the overdue loan and asks them whether they want to pay it back or simply release the loan. If the user selects pay back, then the

TextUI asks the RealEstate object for its Owner and calls TakeOutALien, which we have already defined. The TextUI then continues by releasing the Lien. If the user selects release, the TextUI tells the RealEstate object to release its Lien.

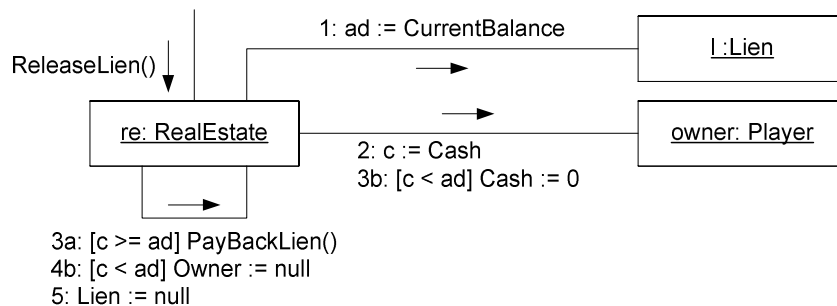
Showing conditional logic on collaboration diagrams can be hard to do well, so here is the equivalent, albeit abbreviated, code for TextUI.LienOverdue(re:RealEstate):

```

1| // This is in TextUI
2| private void LienOverdue(RealEstate re){
3|     int ad = re.LienBalance;
4|     Console.WriteLine("Overdue balance, payback or release");
5|     Console.WriteLine("\tAmount Due: {0}", ad);
6|     string r := Console.ReadLine();
7|     if(r == "pay back") {
8|         Player p := re.Owner;
9|         TakeOutALien(p);
10|    } // end if
11|    re.ReleaseLien()
12| } // end LienOverdue method

```

This is not complete but it shows how we might turn the collaboration diagram into code. We need to do more to handle user input. Also, since we do not yet know what ReleaseLien does, what happens if the user does not get enough cash to pay back the Lien? To know if we are done, we need to see what ReleaseLien might do:



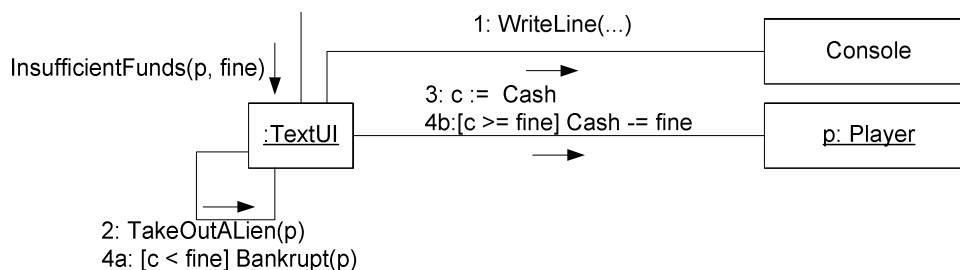
**Figure 48: UML – Collaboration: RealEstate.ReleaseLien**

First get the current Lien balance. Get the Player's cash. If the cash is less than the Lien balance, set the Player's cash to 0 and set the RealEstate's Owner to null. If the Player's cash is greater than or equal to the Lien balance, call PayBackLien. In either case set the RealEstate's Lien object to null.

The combination of these two operations almost gives us what we need. We need to add a bit more to the LienOverdue method to make sure the user has enough cash to pay back the Lien. For now we'll defer that to our code.

### Insufficient funds

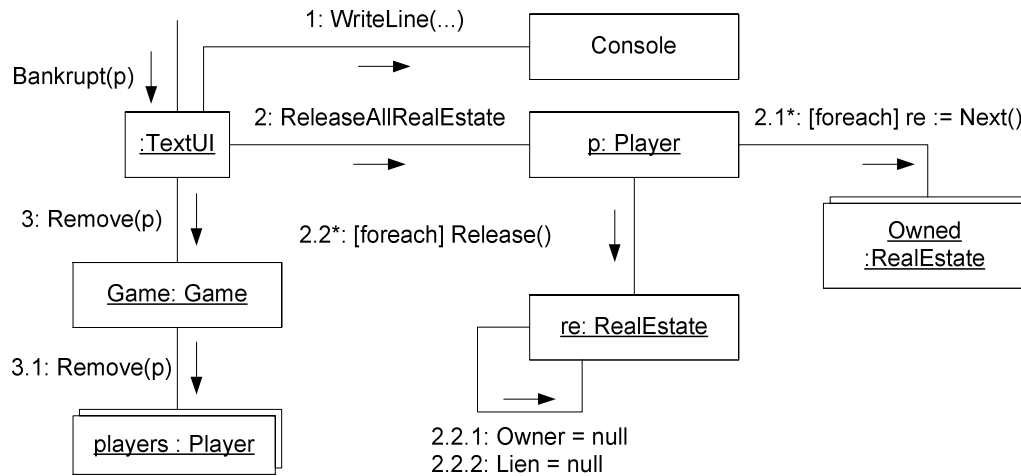
When a Player is charged a fine, they might not have enough cash to pay the fine. When this happens, we need to give a Player the option of getting money to pay the fine or they are out of the game.



**Figure 49: UML – Collaboration: InsufficientFunds(Player, int)**



The TextUI tells the Player it has insufficient funds. It then allows the Player to TakeOutALien. If the Player has enough cash to pay the fine, the TextUI adjusts the Player's balance. If not, the game bankrupts the Player:



**Figure 50: UML – Collaboration: Bankrupt**

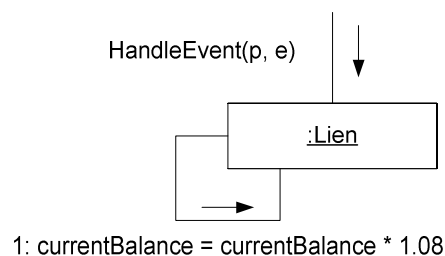
The TextUI informs the user that the Player is bankrupt. It then tells the Player object to release all of its RealEstate objects. Finally, the TextUI tells the Game to remove the player, which the game does.

### Begin turn event

At the beginning of a turn all of the Liens of a Player increase by 8%. This happens before the Player has an option of doing anything to avoid allowing a Player to pay back a lien before the Lien has increased its current balance.

We need Liens to respond to events as we have done with the TextUI responding to begin turn events. We currently have two different places where objects can register themselves as interested in Events, Player object and RealEstate objects. Since RealEstate objects do not currently generate begin turn events, and really should not since they are not involved in the beginning of a turn, the only current option is to use the Player.

Assume we already have handled event wiring, what does this look like:



**Figure 51: UML - Collaboration: Lien.HandleEvent**

This is straightforward, but it does not address three issues:

- The Lien is not wired to respond to events generated by the Player.
- When the Player pays back the Lien or releases a RealEstate object, we need to unwire the Lien object from the Player.
- The Lien will get every broadcast of an event from the Player object.

### **Wiring the Lien**

We can wire the Lien when we create it. If we review Figure 45, we see that the RealEstate object creates a Lien. We can either allow the RealEstate object to wire the Lien to the RealEstate's owner or the RealEstate object can pass in the Player and wire itself. Let's review each of those options:

### **Unwiring the Lien**

Unwiring a Lien happens when either the Lien is paid back or when the Player chooses to not pay back a Lien and the property becomes un-owned. Both of these situations occur in the ReleaseLien method, see Figure 48.

### **Too many Event Broadcasts**

We can address the last issue of the Lien receiving all of the event triggers from the Player in three ways:

- Ignore the issue; a Player does not broadcast too many events so it will not be an issue.
- Reduce the number of event broadcasts within the Player.
- Have some other object broadcast begin turn events and register the Lien with it.

For now we will ignore the problem.

### **Remaining to do's**

Update event triggering: starting (lien updated), moving (overdue thrown), passing, landing, ending

## **Implementing the Design**

We first begin with just taking out a Lien and paying back a lien. After we have addressed those changes, we will move on to the rest of the system operations required by this build.

### **Updates: TurnEventArgs.cs**

```
1| // update the TurnSteps enumeration
2| public enum TurnSteps {
3|     BEGIN, BEGIN_MOVE, OVERDUE, INSUFFICIENT, PASS, LAND, SELL, FINISH
4| } // end TurnSteps enumeration
5|
6| // Rename attribute from myCustomer to myPlayer
7| private Player myPlayer;
8|
9| // Update constructor taking in the player to use new attribute name
10| public TurnEventArgs(string description, Player player, TurnSteps action)
11| : this(description, action) {
12|     Assert.NotNull(player, "player");
13|
14|     myPlayer = player;
15| } // end TurnEventArgs constructor
16|
17| // Replace the Customer property with a Player property
18| public Player Player {
19|     get { return myPlayer; }
20| } // end Player property
```

### **File: Lien.cs**

```
1| using System;
2| namespace BoardGame {
3|     public class Lien {
4|         private int myInitialBalance;
5|         private int myAmount;
6|         private RealEstate myRealEstate;
```

```
7|
8|     public Lien(int amount, RealEstate re) {
9|         Assert.NotNull(re, "re");
10|         Assert.Argument(amount > 0 && amount <= re.Price * 2,
11|             "amount", amount, "Should be 0 to 2x price");
12|
13|         myInitialBalance = amount;
14|         myAmount = amount;
15|         re = myRealEstate;
16|     } // end Lien constructor
17|
18|     public int Amount {
19|         get { return myAmount; }
20|     } // end Amount property
21|
22|     public RealEstate Property {
23|         get { return myRealEstate; }
24|     } // end Property property
25| }
26| }
```

### Updates: Player.cs

```
1| // Simplify the events triggered in Player.TakeATurn
2| public void TakeATurn(PairOfDice dice) {
3|     OnTurn(new TurnEventArgs("Beginning Turn", TurnSteps.BEGIN));
4|     dice.Roll();
5|     Location = Location.Next;
6|     for(int i = 1; i < dice.Value; ++i) {
7|         Location.PassOver(this);
8|         Location = Location.Next;
9|     } // end for loop
10|     Location.LandOn(this);
11|     OnTurn(new TurnEventArgs("Finished Turn", TurnSteps.FINISH));
12| } // end TakeATurn method
```

### File: TextUI.cs

```
1| using System;
2| using System.Collections;
3|
4| namespace BoardGame {
5|     public class TextUI {
6|         private const int DONE = 0;
7|         public void DisplayTurnEvent(object sender, TurnEventArgs e) {
8|             switch(e.Action) {
9|                 case TurnSteps.BEGIN:
10|                     BeginTurnOptions((Player)sender, e);
11|                     break;
12|                 case TurnSteps.FINISH:
13|                     FinishTurn((Player)sender, e);
14|                     break;
15|                 case TurnSteps.PASS:
16|                     Console.WriteLine("\tPassing: {0}", e.Description);
17|                     break;
18|                 case TurnSteps.LAND:
19|                     Console.WriteLine("\tLanding: {0}", e.Description);
20|                     break;
21|                 case TurnSteps.SELL:
22|                     AttemptSale(sender, e);
23|                     break;
24|                 case TurnSteps.BEGIN_MOVE:
25|                     // do nothing
26|                     break;
27|                 case TurnSteps.OVERDUE:
28|                     ProcessOverdueLien((RealEstate)sender, e);
29|                     break;
30|             } // end switch
31|         } // end DisplayTurnEvent
```

```
32 |
33 | private void BeginTurnOptions(Player p, TurnEventArgs e) {
34 |     int result = 0;
35 |     do {
36 |         DisplayBeginTurnOptions(p);
37 |         result = GetNumericSelection(0,2);
38 |         switch(result) {
39 |             case 1:
40 |                 TakeOutALien(p);
41 |                 break;
42 |             case 2:
43 |                 PayBackALien(p);
44 |                 break;
45 |         } // end switch
46 |     } while(result != DONE);
47 | } // end BeginTurnOptions method
48 |
49 | private void DisplayBeginTurnOptions(Player p) {
50 |     Console.WriteLine("{0} you have the following options:", p.Name);
51 |     Console.WriteLine("\t1. Take out a lien.");
52 |     Console.WriteLine("\t2. Pay back a lien.");
53 |     Console.WriteLine("\t0. done with begin turn options");
54 | } // end DisplayBeginTurnOptions method
55 |
56 | private void TakeOutALien(Player p) {
57 |     if(p.OwnedRealEstate.Count == 0) {
58 |         Console.WriteLine("You own no RealEstate objects.");
59 |         return;
60 |     } // end if
61 |
62 |     ArrayList lienFreeRE = new ArrayList();
63 |     foreach(RealEstate re in p.OwnedRealEstate) {
64 |         if(re.HasLien == false) {
65 |             lienFreeRE.Add(re);
66 |         } // end if
67 |     } // end foreach loop
68 |
69 |     if(lienFreeRE.Count == 0) {
70 |         Console.WriteLine("All RealEstate objects already have liens.");
71 |         return;
72 |     } // end if
73 |
74 |     RealEstate selectedRE = GetSelectionFrom(lienFreeRE);
75 |     if(selectedRE == null) {
76 |         Console.WriteLine("Canceling taking out a lien.");
77 |         return;
78 |     } // end if
79 |     Console.WriteLine("Enter Amount up to {0}", selectedRE.Price * 2);
80 |     int lienAmount = GetNumericSelection(0, selectedRE.Price * 2);
81 |     selectedRE.CreateLien(lienAmount);
82 | } // end TakeOutALien method
83 |
84 | private void PayBackALien(Player p) {
85 |     if(p.OwnedRealEstate.Count == 0) {
86 |         Console.WriteLine("You own no RealEstate objects");
87 |         return;
88 |     } // end if
89 |
90 |     IList reWithLien = new ArrayList();
91 |     foreach(RealEstate re in p.OwnedRealEstate) {
92 |         if(re.HasLien == true) {
93 |             reWithLien.Add(re);
94 |         } // end if
95 |     } // end foreach method
96 |
97 |     RealEstate selectedRE = GetSelectionFrom(reWithLien);
98 |     if(selectedRE == null) {
99 |         Console.WriteLine("Canceling paying back a lien.");
```

```

100|         return;
101|     } // end if
102|     selectedRE.PayBackLien();
103| } // end PayBackALien method
104|
105| private RealEstate GetSelectionFrom(ICollection collection) {
106|     DisplayRealEstateSelectionList(collection);
107|     int selectedIndex = GetNumericSelection(0, collection.Count);
108|     if(selectedIndex == 0) {
109|         return null;
110|     } // end if
111|     return (RealEstate)collection[selectedIndex - 1];
112| } // end GetSelectionFrom method
113|
114| private void DisplayRealEstateSelectionList(ICollection collection) {
115|     Console.WriteLine("Please Select RealEstate");
116|     int index = 1;
117|     foreach(RealEstate re in collection) {
118|         Console.WriteLine("\t{0}: {1}", index, re.Name);
119|         ++index;
120|     } // end foreach loop
121|     Console.WriteLine("\t0: quit");
122| } // end DisplayRealEstateSelectionList method
123|
124| private int GetNumericSelection(int low, int high) {
125|     string selection;
126|     do {
127|         Console.WriteLine("\tPlease enter between {0} and {1}", low, high);
128|         selection = Console.ReadLine().Trim();
129|         try {
130|             int result = Int32.Parse(selection);
131|             if(result < low || result > high) {
132|                 Console.WriteLine("\tThe number was out of the range.");
133|                 continue;
134|             } // end if
135|             return result;
136|         } // end try
137|         catch(Exception) {
138|             Console.WriteLine("Didn't understand, please reenter number");
139|         } // end catch
140|     } while(true);
141| } // end GetNumericSelection
142|
143| private void FinishTurn(Player p, TurnEventArgs e) {
144|     Console.WriteLine(
145|         "\t{0} finished turn on {1}", p.Name, p.Location.Name);
146| } // end FinishTurn method
147|
148| private void AttemptSale(object sender, TurnEventArgs e) {
149|     RealEstate re = (RealEstate)sender;
150|     Console.WriteLine("{0} is for sale, purchase it? ", re.Name);
151|     string answer = Console.ReadLine().Trim().ToLower();
152|     if(answer.StartsWith("y")) {
153|         re.SellTo(e.Player);
154|     } // end if
155| } // end AttemptSale method
156| } // end TextUI class
157| } // end BoardGame namespace

```

## Build 6: Planning Events

All kinds of real-estate objects might host an event. Events are a way to temporarily increase revenue. We can develop several complex events but for now we will have XXX events:

- Olympics

- World Cup
- State Fair

Once a player owns a city, they can plan events for the city. An event, such as the Olympics, costs some amount and has some duration. If a player plans an event and pays for it, all players must pay a new amount based on the cost of the event.

## Build 7: Increasing City taxes

### Background: Object Visibility

When we need to design how one object will be able to send a message to another object, we are designing for visibility. There are essentially four ways for one object, in this case Utility to know about another object, in this case PairOfDice:

- Utility has an attribute that holds on to the PairOfDice, (What we chose to do.)
- The object which sent the last message to Utility passed in the PairOfDice object,
- The PairOfDice object is Global,
- The Utility object knows another object through which it can find the PairOfDice object.
- Utility creates its own PairOfDice object.

Imagine someone is trying to get in touch with you. By analogy, that person might:

- Just have your telephone number memorized,
- Might be told your phone number by someone else,
- See your telephone number on a bathroom wall,
- Call information to get your phone number.
- There is no direct analogy to the last option. The closest thing might be some of the US accounting scandals of the early 2000's.

Visibility between objects is just like asking the question, “How can I get in touch with someone later on”? Looking at our five options, here are issues with each:

#### Utility has an Attribute

If Utility has an attribute that refers to the PairOfDice object, it is able to ask for the Value at any time. There are a few issues with that:

- How does the Utility object get the attribute set in the first place?
- Storing an attribute takes up space, is this going to be too costly?
- If there are two utility objects and they share the same PairOfDice as the game and current player, we might have issues with different objects trying to access the same object at the same time. These are so-called race conditions.

A typical time to set an attribute is when creating an object. The BoardLocationReader creates the Utility objects, so it could pass in the PairOfDice when it created the Utility object. This is the solution we have chosen.

#### Utility passed in the PairOfDice

The only time when Utility objects need to use PairOfDice objects is when they calculate taxes. Taxes are only calculated because the Player landed on a Utility that was owned by another player and had no liens on it.

If a player is going to pass in the PairOfDice object to the LandOn method of a Utility, it is straightforward: `current.LandOn(this, dice)`. The player already passes itself in as the first parameter. Now it simply provides another parameter.

When does the player need to pass in the PairOfDice object? Assuming that PairOfDice is the only kind of BoardLocation to use the PairOfDice, then the player only needs to pass in the PairOfDice object when it lands on a Utility. However, the player does not know when it is landing on a Utility object. It only knows that it is landing on a BoardLocation object. The BoardLocation class only has one kind of LandOn method, the one taking a single parameter. Therefore we have to change the LandOn method in BoardLocation, and all other subclasses of BoardLocation to take in the PairOfDice object.

If we count the number of classes we will have in the BoardLocation hierarchy when we finish it, we have 10 classes (including BoardLocation and RealEstate). All 10 versions of LandOn will have to take the PairOfDice object as a parameter, but only one of the 10 classes will use it. Compared to the previous option where we pass in the PairOfDice object to the BoardLocationReader object, the impact of this solution is pretty large.

### **PairOfDice global**

Global objects are not in and of themselves a bad thing. The problem with global objects is that they introduce incidental coupling between different parts of a system. If one part of the system sets a global variable, expecting it to stay the same, and another part changes it, these two parts of the system are coupled to one another in a non-obvious way.

A key problem with the coupling is known as the <<<CHECK>>>Square Law Of Computing, which states[9]:

*The complexity of a problem grows at least as quickly as the square of the size of the problem.*

Global objects increase the size of the problem because the number of ways in which objects can interact becomes much larger. Finding bugs or trying to figure out why a system does what it does becomes much more difficult when we use global variables.

### **Utility knows an object it can ask**

This is more of an advanced option for visibility. Also, this does not solve the problem. While the utility object might know another object that can get the PairOfDice object, it still begs the question, “How did the utility object know this object?” There are 3 answers:

- The object is an attribute of Utility,
- The object is passed in,
- The object is global.

We are right back where we started.

### **Utility creates its own PairOfDice**

In general this is a viable alternative. In this particular case it will not work because the requirements of the system say that the Utility uses the current value of the PairOfDice to determine Tax. If Utility creates its own PairOfDice, there is no connection between the two PairOfDice objects. This will not solve the problem.

Another issue with this is that we will have more objects in memory. This might be what we need so do not rule this alternative out. It also has the advantage of being multi-thread safe. If there are multiple threads in the program but each Utility class has its own PairOfDice object, then we do not have any kind of threading issue because we are not sharing any objects.

## Visibility Summary

When one object, S, needs to send a message to another object, R, the first object must know about the second object. There are many alternatives including:

- S can create its own R every time it needs to send a message,
- S can create its own R once and store it in an attribute,
- Another object that already knows R can tell S about it,
- R can be global,
- S can ask another object is already knows to give it R.

Deciding how one object knows about the existence of another object is called Visibility. The most common forms of visibility are:

- R is passed in as a parameter to a method,
- S has an attribute and it is provided this attribute at creation time.

Making choices about visibility is a major topic of the GRASP patterns in [4].



## Iteration 3a: ADO.NET

Define a simple table structure.

Read it.

Provide a class that will read from and write to the class.

Do everything without web matrix.

[Estimate 3 pages for setup and configuration – as an appendix]

[Estimate 15 pages for whole thing.]

### **A Problem**

### **A Solution**

#### **Code**

#### **Code Explained**

#### **Build & Execute Instructions**

#### **Summary**

## **Iteration 3b: Simple UI**

25 pages.

**A Problem**

**A Standalone Example?**

**A Solution**

**Summary**

## **Iteration 4: Remaining Classes**

**A Problem**

**A Solution**

**Summary**

## Appendix: Visual Notation

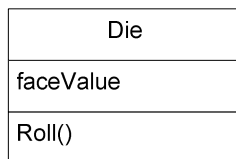
Throughout these examples we will be using the Unified Modeling language. For a complete reference, see [7]. This section is a quick summary. We discuss static notation and dynamic notation as we are using it throughout this book.

Note that we are not trying to present a comprehensive description of UML notation. We are also using it in a specific way. We will define interpretations for the notation that, while acceptable, are not the whole story. To quote [2], we are using lies to adults.

### Static Notation

Static notation shows the parts that make up a system without showing the order in which they collaborate.

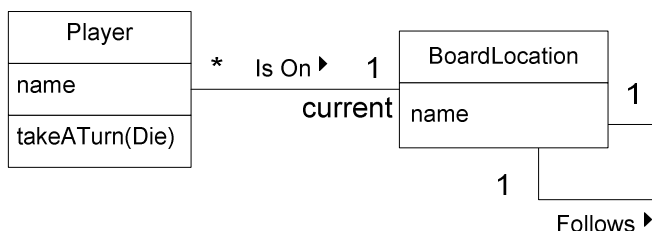
#### The Class



**Figure 52: A Class**

This is the symbol to represent a class. This may already be familiar to you. In this case the box has three compartments. The top compartment is the name of the class. The middle compartment lists attributes or properties. The bottom part lists methods.

#### Association



**Figure 53: Associations**

The solid lines between classes are associations. An association shows that two classes are somehow related. In this example we have two associations:

- Player objects are associated with BoardLocation objects. The association is known as “Is On”.
- A Board Location is associated with another Board Location. The association is known as “Follows”.

There are two more things to mention in this example. First, there are the following symbols: \*, 1. Second, there is the word “current.” The symbols refer to multiplicity. The word current is known as a role.

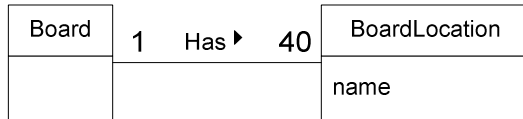
### Multiplicity

Multiplicity says how many of one thing is related to how many of another. We can read the association in both directions. We can read the first association as follows:

- One Player “Is On” one Board Location.

- One Board Location has \* (zero or more) Players on it.

Notice that when reading from left to right, Player first, we said “one” Player. The same is true when we read the relationship in the reverse direction. This is a convention for reading multiplicity. As another example, imagine the following association between two classes:



**Figure 54: Multiplicity**

Reading from left to right, “One Board has 40 Board Locations.” Reading from right to left, “One Board Location is on one Board.”

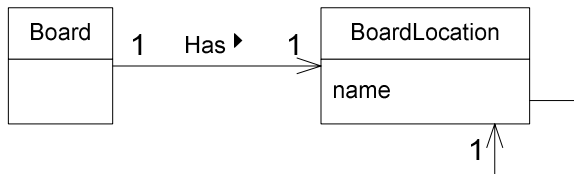
The ◀ ▶ ▲ ▼ represent the direction in which to read the word. For example, “Has ▶” means that reading from left to right, use the word “Has”. Reading from right to left it means “insert some meaningful replacement to make what you say make sense”.

## Role

The word “current” on the line representing an association is something to give a little clarification to the association. In the example above, the Player “Is On” one Board Location and that Board Location is known as “current”. That is, the player’s current location is exactly 1 board location.

## Navigable

An association between classes might be further refined as follows:



**Figure 55: Navigable**

The arrowhead on a solid line states that we can navigate from the board to one BoardLocation object. The BoardLocation objects do not know about the board. This is a decision we make during design.

The line from the BoardLocation back to itself indicates that a BoardLocation object knows about one other BoardLocation object.

## Dependency

We will use one final bit of notation to show that one class somehow depends on another class:



**Figure 56: Dependency**

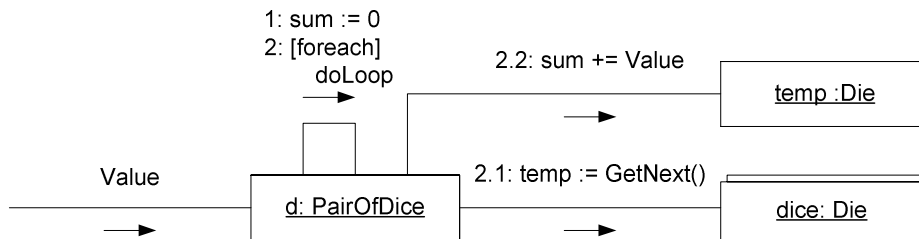
The arrowhead on a dashed line means that one thing depends on another. The dependency probably means “uses in some way.”

## Dynamic Notation

Dynamic notation, as its name implies, shows the order in which the parts that make up the system interact. There are two primary diagrams we use, sequence diagrams and collaboration diagrams. We

will use collaboration diagrams exclusively for detailed interaction. We will use sequence diagrams exclusively for system level interaction or iteration between layers within a system.

## Collaboration Diagram



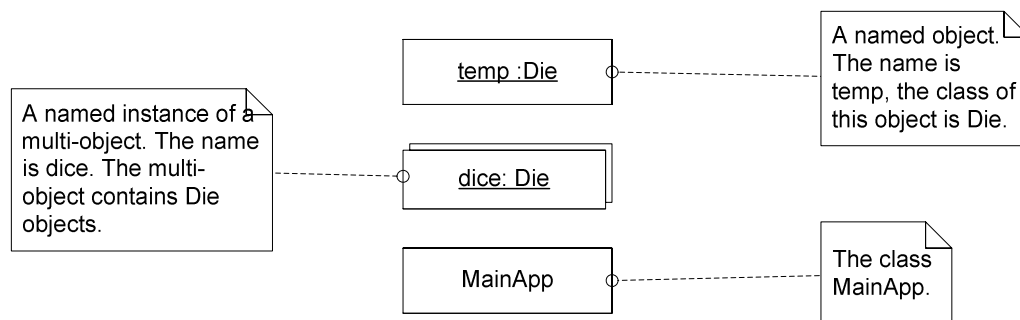
**Figure 57: Collaboration: Example**

A collaboration diagrams shows the interaction between instances or sometimes between classes and instances. Furthermore, these diagrams use a hierarchical numbering system so we can tell the order in which everything occurs.

By convention, the message that starts everything off is not numbered. In this example, Value is the message that starts everything off. Next, the following things happen:

- Create a temporary variable called sum and set it to 0.
- Begin a loop
  - For step one of the loop, acquire an object from a collection of objects.
  - For step two, send that newly acquired object the message Value and add the resulting value into the sum.

## Classes & Objects



**Figure 58: Collaboration: Instances and Classes**

We will see three kinds of boxes on these diagrams. The first, and most common, is an instance. In this diagram, we show an instance of the Die class. We know it is an instance of a Die because of the “:Die”. In general, :Name indicates “instance of this kind of class”.

The second item is a multi-object. A multi-object represents a grouping of objects. When we write code using a multi-object, we might use an array or a collection class. We use this notation to represent both. In this case, the “:Die” means that the multi-object contains Die objects.

The final box is missing the underline. When we see a box like this on a collaboration diagram, the underline indicates object or instance of a class. When the underline is missing, it means class. When we use a static method (also known as class method), we will see a box without an underline,

indicating that this is a message to the class, not to an object. The most common example we will see of this is `MainApp.Main()`. The `Main()` method is a static method, which means it can be called without having to first create an instance of the `MainApp` class.

### **Numbering**

The numbering system is hierarchical. From this example we can see that steps 2.1 and 2.2 are part of step 2. Since step 2 is a loop (by convention we use loop), it follows that the first line of the loop is to acquire an object out of a multi-object. Step 2 is to send a message to that object.

### **Guard**

The square brackets, `[]`, indicate that we are doing something as long as some condition is true. In this case we use the word “foreach” to indicate that we will do this loop for every element in the multi-object.

<<<Review all collaboration diagrams – finish this section based on what is in there>>>

# Appendix: Pre versus Post Increment

## Overview

In “for” loops, you will see throughout this and all of our examples that we use pre increment, `++i`, instead of post increment, `i++`. In the case of C# this does not make a difference. If you have used C++, and specifically used the template collections in the standard library, you might have come across this. If you have not come across this and are curious, read this section.

## It is the same

First, what does a for loop really do? We will rewrite a typical for loop in terms of its equivalent while loop form.

```
for(int i = 0; i < 100; i++) {  
    // do something  
}  
// becomes  
int i = 0;  
while(i < 100) {  
    // do something  
    i++;  
}
```

The increment of the loop variable happens as the last step, and it happens by itself. Looking at this rewrite, it is hopefully clear that using `++i` versus `i++` makes no difference.

## What is the difference

The difference is in the result of the evaluating of the expression. Suppose we show what the two expressions would look like if they were functions.

### Pre Increment as a Function

```
ref int PreIncrement(int i) {  
    i = i + 1;  
    return i;  
}
```

### Post Increment as a function

```
int i PostIncrement(int i) {  
    int copy = i;  
    i = i + 1;  
    return copy;  
}
```

In the case of pre-increment, `++i`, we get back the original variable. What we get back is called an L-Value. It is called an L-Value because it can appear on the LEFT hand side of an assignment statement. On the other hand, post-increment, `i++`, results in a copied value being returned.

This really does not make any difference in C# but it does make a difference in C++, especially when working with the iterators built in to the standard library.

## Conclusion

Using pre-increment in for loops does not make any difference in C#. It can make a difference in other languages. So the author chooses to use the form that in most cases makes no difference and in some cases actually can provide improved performance.

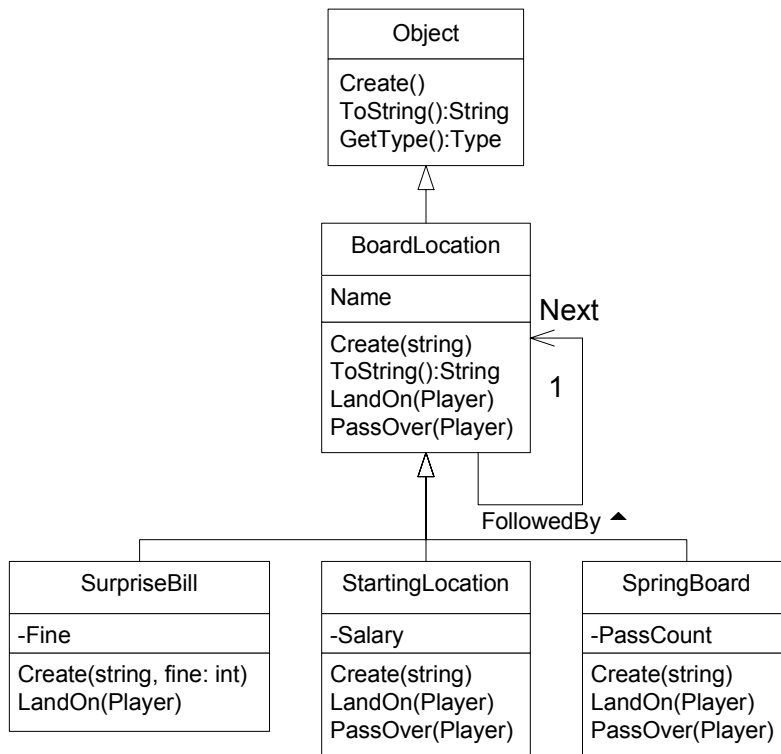


## Appendix: Virtual Methods Explained

When one object sends a message to another object, the system maps the messages to a method. In the case of non-virtual methods, this mapping happens before the program is executed. If the method is virtual, the mapping happens during runtime.

As of the second iteration of our Game example, Player objects use two virtual methods of the BoardLocation class. These methods are: LandOn(Player) and PassOver(Player).

The total BoardLocation hierarchy for iteration 2 is:



**Figure 59: UML – Structure: BoardLocation Hierarchy**

Notice that every class in this hierarchy, and in C# for that matter, inherits either directly or indirectly from the **Object** class. Each object has a type, or a class from which it was instantiated. An object knows its class and we can use the `GetType` method to get the type of an object. A class knows its virtual methods. (This is actually a lie to adults, but it is close enough for the current discussion.)

Imagine a smaller board with two **BoardLocations**, one **SpringBoard**, one **SurpriseBill** and one **StartingLocation**. This is a partial snapshot of what memory looks like, including the class objects, references to virtual tables and references to the code for each method. Note that we are not showing all virtual methods, just the ones we use. Also, we do not show that each class knows its super class.

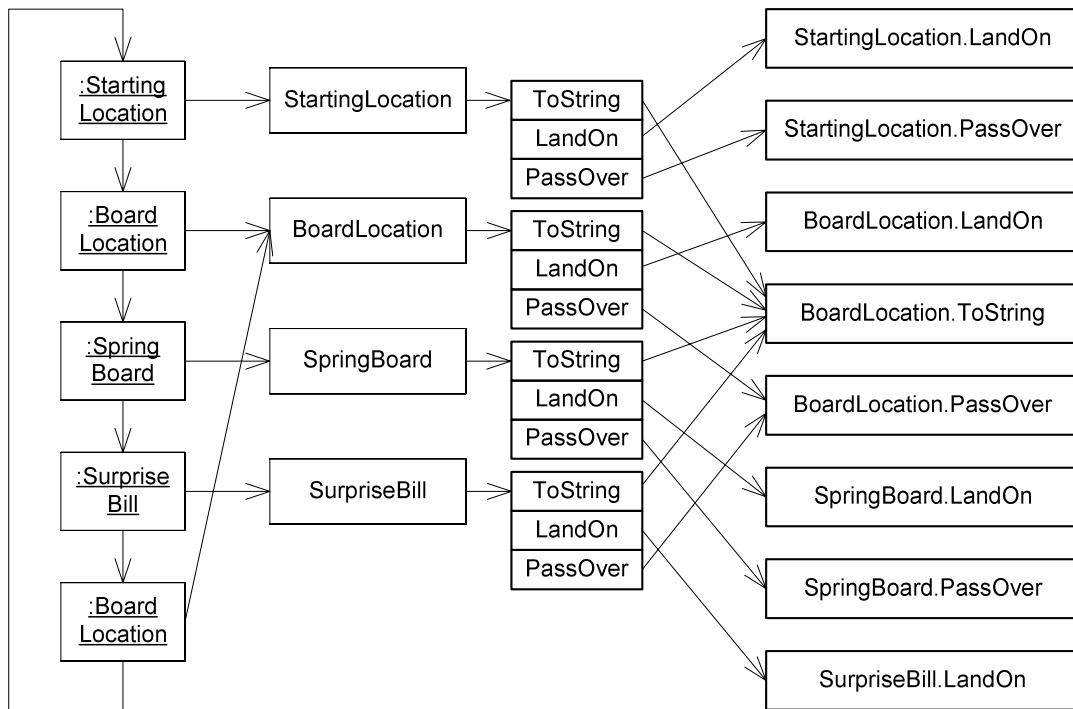


Figure 60: Virtual Function Tables

## Follow the Chain

### Object → Class

When some part of the code attempts to execute a virtual method, the first step is to find the associated class object:

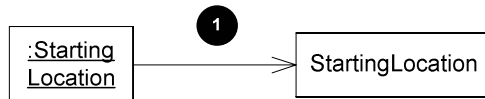


Figure 61: Object → Class

Each object knows its class. Both BoardLocation objects point to the same class object. An instance of SpringBoard points to the SpringBoard class object. The SurpriseBill points to the SurpriseBill class object. And finally, the StartingLocation object points to the StartingLocation class object.

### Class → Virtual Method Table

Each class knows all of its virtual methods. Notice that each of the classes shows three virtual methods, again there are more, we are only showing a partial list. Each class has a table of all of its virtual methods. Also notice that the order is the same in all cases. Finally, virtual methods defined in the base class are listed before virtual methods listed in the derived class.

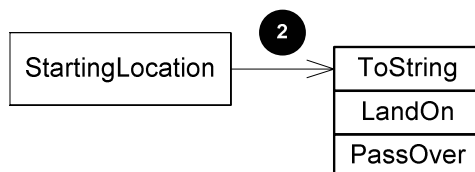
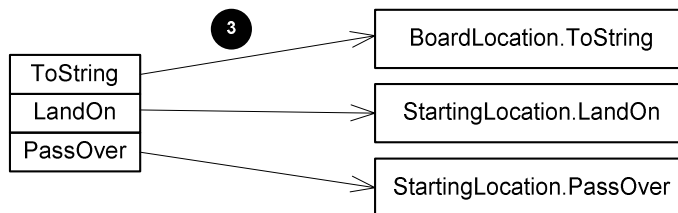


Figure 62: Class → Virtual Method Table

Each of these table entries are references to the actual code that executes when a method is invoked. There are a few things worthy of note. First, there is only one version of ToString and it is shared by all of the class objects. We write a version of ToString in the BoardLocation class, all of the subclasses inherit it as is. StartingLocation replaces both LandOn and PassOver, so we see that its table of virtual methods points to two unique bodies of code for each of those methods.

### Virtual Method Table → Code

Finally, C# will follow the reference in the virtual method table to the actual code for the virtual method.



**Figure 63: Virtual Method Table → Code**

### Lookup

When we attempt to send a message to an object, C# first determines if the message is or is not virtual. If the method is not virtual, C# is able to connect the message to the method before the program actually executes.

If the method is virtual, then C# will instead insert an op-code to perform the lookup at runtime. The lookup is based on two things: the “this” pointer and the offset into the virtual method table of the method. The reason the order of entries in the virtual table is important is to allow this pre-computing to occur. The methods are always listed in the same order and the base methods are before the derived methods. C# is able to calculate the correct method location at runtime by performing a single simple integer pointer addition to an address. The virtual table is always in the same relative location from the address of the “this” pointer. The overhead to call a virtual method is very small.

### ToString

In the second iteration of the Game application, we had a simple ToString method:

```
1| public override string ToString() {
2|     return GetType().Name + "(" + Name + ")";
3| } // end ToString method
```

Imagine the following code example:

```
4| BoardLocation current = new BoardLocation("Name1");
5| Console.WriteLine(current.ToString());
6| current = new StartingLocation("Name2");
7| Console.WriteLine(current.ToString());
```

In line 5, we send the ToString() message to current.

- C# determines that current is an instance of the BoardLocation class
- C# goes to the virtual table for BoardLocation
- C# performs a simple indexing operation and finds the pointer to the ToString method used in BoardLocation.
- C# executes the method.

Within the method, the “this” pointer points to the same object that current points to, namely the object created on line 4. The GetType() message is sent to the “this” object and it returns the class object representing BoardLocation (technically the class of the return object is “Type”). When we ask

this object for its Name property, it returns “BoardLocation”. We then concatenate “BoardLocation” + (“ + “Name1” + “)” to end up with “BoardLocation(Name1)”.

On line 6, we reassign the reference named current to a new object. This time it is an instance of the class StartingLocation. In line 7 we send the ToString() message to the object referred to by current. In the ToString() method, we use the GetType() method. The “this” pointer now points to the same object that was created on line 6. The Type object returned is the one that represents the StartingLocation class. When we ask for its name we get “StartingLocation”. We concatenate to get “StartingLocation(Name2)”.

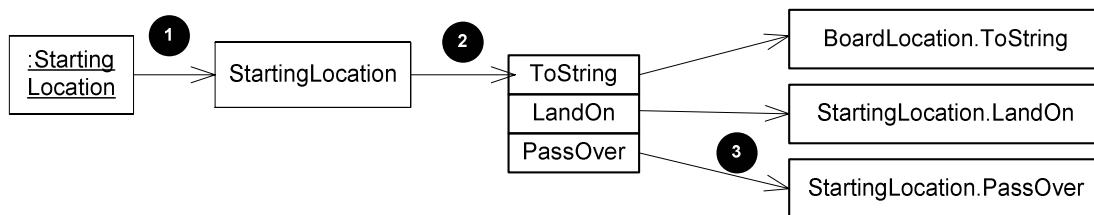
## LandOn

The same thing is happening with LandOn. The only difference is that several subclasses of the BoardLocation class replace the base version, so we get different responses based on the class.

Consider the following code example from Player.TakeATurn:

```
1| public void TakeATurn(PairOfDice dice) {
2|     // some code removed
3|     myLocation = Location.Next;
4|     for(int i = 1; i < dice.Value; ++i) {
5|         myLocation.PassOver(this);
6|         myLocation = Location.Next;
7|     } // end for loop
8|     myLocation.LandOn(this)
9| } // end TakeATurn method
```

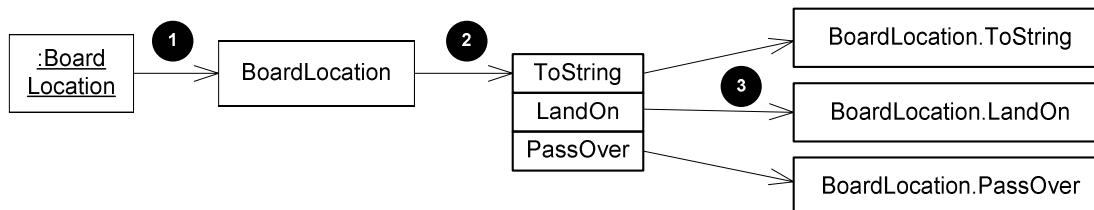
In lines 5 and 8, the player sends a message to a BoardLocation object. Remember that a message is the request; the method is what the system ultimately executes. Since these methods are virtual, C# must perform the indexed lookup at runtime to select the correct method. Assume the player passes over the StartingLocation:



**Figure 64: Player Passes Over StartingLocation**

In line 5, the player issues a message that is virtual. C# will use the address of the `StartingLocation` object plus some constant value (the offset from the beginning of the object to its class object), add to that (using pointer addition) the offset of `PassOver` in the virtual table (in this example it is 2, in reality it is more like 7) and finally execute the code that it finds.

Later, the player finally lands on a `BoardLocation`:



**Figure 65: Player Lands On a BoardLocation**

The process is the same. Use the address of the object to which the Player sends the `LandOn` message. Add some constant value to find the class object. Find the virtual table in the class object (it is at a well-defined place), use pointer arithmetic adding the offset of `LandOn` into the virtual table, find the method and execute it.

In both examples, C# will execute the method and do two things. First, it will pass the address of the receiving object, a BoardLocation object, as the first parameter. This parameter is called the “this” pointer. Second, it will pass any parameters provided when sending the message. In this case, the player passes itself. For both LandOn and PassOver, there are two actual parameters. The “this” pointer and a reference to the player.

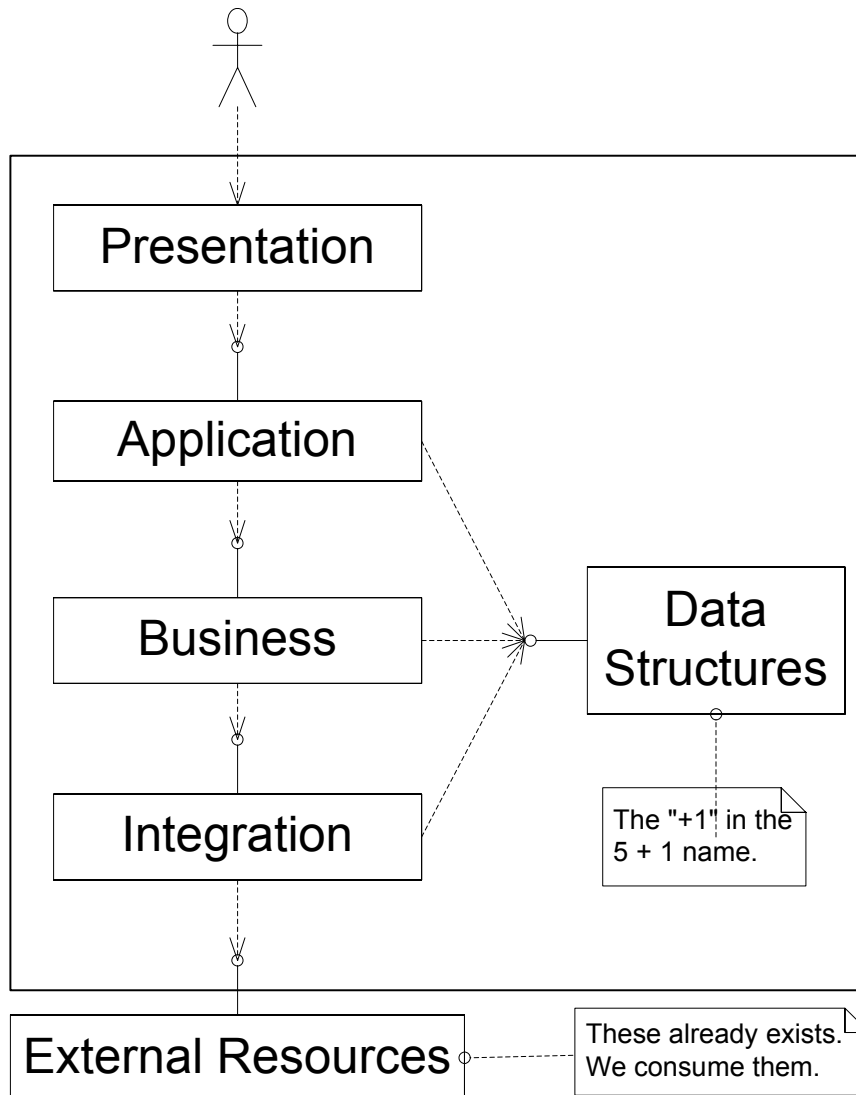
## Appendix: Summary of the 5 + 1 Layers Pattern

Software systems have some overarching structure to them. There are several reoccurring styles of organization that seem to have survived through natural selection. One of these patterns is the Layer's pattern, introduced by XXX. See [8] for a description of that pattern.

The layers pattern describes a system as comprising several different logical layers. This section provides on possible list of layers you might consider when developing your systems. The layers we will discuss are:

Layer	Description
Presentation	What a users sees and how that user interacts with the system.
Application	The order in which a user uses the system. Different users might use the system in different ways. The application layer determines the particulars for a given user.
Business	The rules of the system. This is where most of the functional requirements end up
Integration	Connectivity to anything outside of the system. Accessing files, databases or other legacy systems will happen in the integration layer.
External Resources	These are the actual resources outside of the system. These are not really a layer within the system but rather just outside of the system. We include this to make the picture more complete.
Data Structures	This is not actually a layer, it is the +1 in the name. Typically there are common data objects that tie the application, business and integration layers together.

Here is a picture showing the typical relationship between these layers:

**Figure 66: 5 + 1 Layer View**

## Presentation

The presentation layer enables a user to interact with the overall system in some manner. The presentation layer may be implemented by multiple subsystems, e.g. one presenting a textual interface and one presenting a GUI.

The presentation layer:

- Contains User Interfacing (presentation) logic only.
- Does not contain application/workflow or business logic.
- May physically reside either on a client machine or on a server machine or both.

The presentation layer represents the V in MVC or Model-View-Control. A presentation layer may be implemented using several technologies including:

- GUI components, e.g. Swing or MFC
- Textual user interface
- Web-based presentation

We do not implement business logic in the presentation layer. One way to accomplish this is to disallow access to the domain objects within the presentation layer. The presentation layer relies on the Application layer to map between the presentation layer and the Business layer and take care of converting non-domain objects into something the interfaces of the Business layer can consume.

## Application

The Application layer implements application workflow. It contains the controller part of the Model View Control pattern. It maps requests from the presentation layer into appropriate requests of the Business layer. It also manages the flow and control of the user's experience.

Robotic actors are likely to not use either a presentation or Application layer but instead sit on top of the Business layer<sup>2</sup>.

The Application layer coordinates the efforts of the presentation layer and converts from presentation layer representations to Business layer representations. In effect, the Application layer converts between a "flat" representation and a "domain object" representation.

The Application layer may be implemented using several techniques:

- Custom code
- Biztalk

## Business

The Business layer represents a unified set of business components implementing functionality required by client applications.

This layer:

- Provides the application controllers from the Application layer with interfaces that execute specific business tasks (e.g. Business Transactions).
- May or may not implement actual business rules. For any legacy systems, the Business layer normalizes the interfaces to something contemporary clients can easily work with. If the application aggregates new and existing functionality, the Business layer contains new business logic but delegates existing functionality requests to the host's legacy API/service via the Integration Layer. From the client's perspective, this layer implements all business functionality.
- Hides the complexity of calling legacy systems or distributed APIs/services.

The Business layer presents business functionality in the form of either a Façade or several "smaller Façades" or components. The interfaces of these components typically contain a small number of methods (e.g.  $7 \pm 2$ ) that are coarse-grained and functionally oriented.

The Business layer may be implemented using several technologies:

- EJB/J2EE
- COM+
- .NET
- RMI
- RPC

A typical interface contains methods grouped according to several dimension of cohesion including:

- Same actor consumes

---

<sup>2</sup> This moves towards more of a B2B configuration. When one business consumes the Business layer of another company over the Internet, this becomes a web services configuration.



- Part of a logical business process
- Happen together close in time/logically follow
- Impact same set of data
- Are intuitively related somehow

We might split a component for various reasons:

- Different security constraints
- Grouping requires unnecessary statefulness
- Too complex

Our service components should have their distribution and technology hidden by providing a Translation Proxy<sup>3</sup> for clients.

Note that even though we are hiding their distribution and technology components, their interface methods are written to a granularity appropriate for the planned context of usage. We have seen more projects fail or get into serious trouble because they did not consider physical distribution during interface design.

## Integration

This layer:

- Contains components that supply technical services,
- Encapsulates the source (platform, technology, databases, etc.) of the services and information.
- Hides the location of the external resources,
- Normalizes the interfaces of disparate systems.

The Integration Layer wraps existing functionality, external functionality, etc. It might use any number of technologies including:

- Screen scraping
- TCP/IP
- Various MOMs
- RPC
- Other custom RPC mechanisms
- So-called connectors

Having wrapped external functionality, the Integration Layer presents to the Business Layer an interface that maps more closely to how the Business Layer perceives the world.

The Integration Layer also hides the location of external resources.

## External Resources

External Resources comprise external systems used in the implementation of the system. This might include:

- SAAS, Oracle, Image, etc.
- Existing legacy components
- External web services

---

<sup>3</sup> AKA Rebinding Proxy.

We include this layer for two reasons:

- It provides a way to model the complete system and demonstrate what kind of functionalities, interfaces, protocols, etc. the system will have to manage.
- We might have to implement some kind of configuration, scripting or other content in the management of these External Resources.

External Resources are things we plan to use in our system but which are generally out of our control. These things exist in different forms and different locations. The Integration Layer normalizes all external resources.

## Data Structures

The data structures “layer” is not really a layer. The Application, Business and Integration Layers consume it.

These objects are best described as data structures. They implement only intrinsic functionality. Extrinsic functionality resides in the Business layer.

The Domain Layer represents the “+1” in the name.

We use the term Layer here loosely. In the original layers pattern, there was a strict separation of layers. A given layer talked to layers below it and was used only by layers above it. In our case, we have common data structures living across logical layers.

Objects in this layer should only implement intrinsic functionality. Composite business rules exist in the Business Layer.

As we develop a Domain Layer, we have to consider whether we care about the Domain Layer for the current project only or for multiple projects. For example, one set of requirements suggests a “one to many relationship” between two objects another set of Requirements for similar objects suggest a more restrictive multiplicity, such as one to one or one to three.

You have a few options:

- Assume this will not happen and do not worry about it (this is a very likely scenario, if you can get away with it, use it).
- Implement the more flexible mechanism and use additional business rules outside of the data structure, say at the business component layer.
- Use program to interface and provide multiple implementations of the data structures.
- Externalize the rule validation and plug in an appropriate validation algorithm.
- Re-implement.

It is more important to be aware that this might happen. The first two options above are going to cover most of the situations that arise.

## Exceptions across Layers

We hide exceptions generated by technologies such as the distribution mechanism (CORBA, EJB, RMI, etc.), database, etc. We create various “Application Exceptions” such as “ServiceNotAvailable.” We might distinguish between:

- Not likely a retry will work.
- Retry possible.

Given the use of Translation Proxies, we might already implement retry logic so that by the time we get a system-level exception we know we are not going to succeed right now

## References

- [1] OMG UML Standard
- [2] Rational Definition
- [3] Science of Disc-World, Terry Prachett
- [4] Applying UML with Patterns, Craig Larman.
- [5] Robert Martin Paper on Packaging
- [6] Object-Oriented Software Construction, Bertrand Meyer
- [7] OMG UML Reference 1.4
- [8] Reference to the Layer's paper.
- [9] An Introduction to General Systems Thinking, Gerald M. Weinberg
- [10] Design Patterns, Gamma Et Al.